

Devpac 3
for the Amiga

USER MANUAL



Devpac 3

for the Amiga



The Old School, Greenfield, Bedford MK45 5DE, UK
tel +44 (0) 1525 718181 • fax +44 (0) 1525 713716
www.hisoft.co.uk • www.cinema4d.com

Devpac 3 for the Amiga®

By HiSoft

© Copyright 1991 HiSoft. All rights reserved.

Program: designed and programmed by HiSoft

Manual: written by David Nutkins, Alex Kiernan and Keith Wilson

This guide and the Devpac 3 program diskettes contain proprietary information which is protected by copyright. No part of the software or the documentation may be reproduced, transcribed, stored in a retrieval system, translated into any language or transmitted in any form without express prior written consent of the publisher and copyright holder(s).

HiSoft shall not be liable for errors contained in the software or the documentation or for incidental or consequential damages in connection with the furnishing, performance or use of the software or the documentation.

HiSoft reserves the right to revise the software and/or the documentation from time to time and to make changes in the content thereof without the obligation to notify any person of such changes.



Published by HiSoft

The Old School, Greenfield, Bedford MK45 5DE UK

First Edition, November 1991 - ISBN 0 948517 47 6

Reprinted with minor corrections - June 1992

Table of Contents

Preface to Devpac 3	1
Introduction	1
How to use the Manual	2
A Course for the Beginner	2
A Course for Seasoned Assembler Programmers	3
Devpac Version 2 Users	3
System Requirements	4
Typography	4
Typefaces	4
Acknowledgements	5
A Quick Tutorial	6
Chapter 1 Introduction	11
Devpac 3 Disk Contents	11
Program Disk	12
Include disk	13
Always make a back-up	14
Installation	14
Registration Card	15
The README File	15

Chapter 2 Using the Editor **17**

Introduction **17**

The Editor **18**

Starting the Editor **19**

Requesters and gadgets **20**

Menus and sub-menus **26**

The Editor's windows **28**

Entering text and moving the cursor **31**

Bookmarks **34**

Deleting text **35**

Block Commands **36**

Searching **39**

Disk Operations **42**

Macros **45**

Settings **47**

Assembler settings **51**

Miscellaneous Commands **53**

Assembling Programs **55**

Program menu **57**

Error commands **59**

Keyboard command summary **61**

Chapter 3 The Assembler	65
Introduction	65
Invoking the Assembler	65
From the Editor	65
Assembly to Memory	71
Stand-Alone Assembler	71
Assembly Process	74
Return Codes	75
Binary file types	75
Types of code	77
Assembler Statement Format	78
Label field	78
Mnemonic Field	79
Operand Field	79
Comment Field	79
Expressions	80
Local Labels	89
Instruction Set	91
Word Alignment	91
Assembler Directives	93
Assembly Control	93
Assembler Directives	106
Repeat Loops	110
Listing Control	110
Label Directives	113
Floating Point Directives	117
Conditional Assembly	119

Macro Operations	122
Output File Formats	130
<i>Choosing the Right File Format</i>	131
Output File Directives	131
<i>Sections</i>	131
<i>Imports & Exports</i>	133
<i>Motorola S-records (SREC L6)</i>	135
Directive Summary	136
Chapter 4 The Debugger	139
<i>Introduction</i>	139
MonAm Concepts	140
<i>Exceptions</i>	140
<i>Front Panel Display</i>	142
<i>MonAm and Multi-Tasking</i>	145
<i>Symbolic Debugging</i>	145
<i>MonAm Requesters</i>	146
<i>Command Input</i>	147
MonAm Overview	147
MonAm Reference	151
<i>Numeric Expressions</i>	151
<i>Window Types</i>	154
<i>Cursor Keys</i>	158
<i>Window Commands</i>	159
<i>Other A Commands</i>	164
<i>Screen Switching</i>	165

Breakpoints	166
<i>History</i>	169
<i>Quitting MonAm</i>	170
<i>Loading & Saving</i>	171
<i>Executing Programs</i>	172
Searching Memory	173
Miscellaneous	175
Command Summary	180
Debugging Stratagem	182
<i>Restrictions</i>	182
<i>Hunting</i>	182
<i>Exception Analysis</i>	183
 Chapter 5 The Linker	 187
A simple Blink command line	187
Concepts	187
<i>ALVs</i>	188
<i>Near DATA/BSS</i>	188
Directives	188
<i>Input directives</i>	189
<i>Output directives</i>	189
Map files	191
Options	192
WITH' files	193
Special HUNK names	195
Reserved symbols	195

Blink Messages	196
<i>Blink Warnings/messages</i>	196
<i>Blink Errors</i>	197
 Chapter 6 Other Tools	 205
S-record Splitter	205
<i>Command line examples</i>	206
Operating system utility	206
<i>FD2LVO details</i>	207
 Appendix A AmigaDOS Error Codes	 209
 Appendix B GenAm Error Messages	 213
Errors	213
Warnings	220
 Appendix C Calling the System	 223
Introduction	223
Libraries	223
<i>Disk font Library</i>	225
<i>DOS Library</i>	225
<i>Exec Library</i>	226
<i>Graphics Library</i>	226
<i>Icon Library</i>	226
<i>Intuition Library</i>	227
<i>Maths Libraries</i>	227
<i>Release 2.0 libraries</i>	227

Example Programs	229
<i>demo.s</i>	229
<i>freemem.s</i>	229
<i>helloworld.s</i>	229
CLI vs Workbench	230
<i>CLI Startup</i>	230
<i>Workbench Startup</i>	230
Other 68000 Series Processors	231
 Appendix D Using the CLI	 233
<i>Introduction</i>	233
<i>Files, Volumes and Directories</i>	233
<i>AmigaDOS Wildcards</i>	235
<i>Device Names</i>	235
<i>AmigaDOS Commands</i>	237
<i>Startup Sequence</i>	244
 Appendix E Floating Point Co-processor	 245
<i>Extended precision</i>	245
<i>Double precision</i>	246
<i>Single Precision</i>	246
<i>Packed Decimal</i>	247
<i>FPCR Floating point control register</i>	248
<i>FPSR Floating point status register</i>	249
<i>MAR Floating point instruction address register</i>	250

Appendix F New Features	251
Summary of Version 3 Improvements	251
<i>The Editor</i>	251
<i>The Assembler</i>	252
<i>The Debugger</i>	253
<i>Integration</i>	254
<i>Linker</i>	254
<i>New include files</i>	254
<i>Features added to Devpac Amiga version 2</i>	254
 Appendix G Converting Programs	 257
<i>Amiga® Macro Assembler and MCC Assembler</i>	257
<i>K-Seka</i>	258
<i>Assempro</i>	258
<i>ArgAsm</i>	258
 Appendix H Technical Support	 259
 Appendix I Bibliography	 261
<i>Amiga</i>	261
<i>680x0</i>	262
<i>Algorithms & Data Structures</i>	264

Preface to HiSoft Devpac 3

Introduction

HiSoft Devpac 3 (called simply HiSoft Devpac from now on) is a complete package for the production of fast, efficient assembly language programs on your Amiga® computer.

There is an *editor* for the creation and editing of your assembler source code, a *linker* for building your programs together with other object files, a *debugger* for helping you to stamp out those nasty bugs (problems) and, of course, an *assembler* to turn your source code into speedy, compact machine code.

This chapter is an introduction to this manual which aims to cover all aspects of installing and using HiSoft Devpac on your Amiga® computer - it does not attempt to teach you 680x0 programming although the accompanying 68000 pocket book and the examples should be of assistance in this regard. For further reading, you should consult the *Bibliography*. You may wish to take advantage of our special offers on a range of technical books that you will find invaluable if you are a serious assembler language programmer; for details, see the order form included with this package.

Please spend some time and effort getting to know and learning how to use the manual so that you can gain the maximum benefit from HiSoft Devpac.

The rest of this section explains how to use the manual, whether you are a beginner or an expert, how to use your computer to best effect with HiSoft Devpac and, finally, we outline the different type styles that we have used throughout the manual to (hopefully) make it easy and enjoyable to use.

How to use the Manual

We have designed this manual to tell you about using HiSoft Devpac on the Amiga® computers. We have packed a great deal of information about the package into the manual and, in order to help you use it efficiently and easily, we will now plot recommended courses through the manual for you, whether you are a beginner to BASIC or a seasoned expert.

A Course for the Beginner

If you are a newcomer to assembly language then we recommend that you read one of the books in the Bibliography alongside this manual.

At the end of this preface there is a simple tutorial which you should follow to familiarise yourself with the use of the main parts of the program suite.

Chapter 1 is an introduction to using Devpac and covers the contents of your master disk, making a back-up copy of it, installing Devpac and registering your purchase.

Chapter 2 considers the editing environment with an overview of using the package and is well worth reading; much of *Chapter 3*, detailing the assembler, is liable to mean little until you become more experienced but should be used as a reference. The overview of the debugger in *Chapter 4* is recommended, though the detail of this package can be left for a while. *Chapters 5* and *6* can be omitted unless you are linking your programs together or using SRecords. Looking at and running the supplied source code should be helpful.

The Appendices are mainly for reference and you will only need to dip into them occasionally.

We hope you find HiSoft Devpac easy and friendly to use, please do not hesitate to write to us with any suggestions for improvements and/or alterations.

A Course for Seasoned Assembler Programmers

If you are experienced in the use of 680x0 assembly language but have not used a member of the Devpac family on the Amiga® before then here is a very quick way of assembling a source file:

Load Devpac, Press **AO** and select your file which will load into the editor. Use **Ctrl-1**, **Ctrl-2** and **Ctrl-3** to access the assembly options requesters and select the options which you require. If generating executable code then select **Assemble to Memory** on the Settings menu for additional speed. Pressing **Ctrl-A** will start the assembler.

Any assembly errors will be remembered and on return to the editor you will be placed on the first one. Subsequent errors may be found by pressing **Ctrl-E**.

To run your successfully-assembled program from memory or disk, press **Ctrl-X**.

As a quick introduction to the debugger the example at the end of this preface is recommended. If you have any problems *please* read the relevant section of the manual before contacting us for technical support.

The Appendices are for general reference and it is worth glancing through all of them to acquaint yourself with their contents.

Good luck, we hope you find HiSoft Devpac a powerful, flexible and easy-to-use development system. Of course, we welcome any written comments you may have on how we might improve both the program and the manual.

Devpac Version 2 Users

Turn to *Appendix F* and read the section summarising the new features, then read *Chapter 2* which covers the editor. The beginning of *Chapter 3* covers the new assembly options.

System Requirements

HiSoft Devpac will run on any Amiga® 680x0 computer (A500, A1000, A2000, A3000 etc.) with at least 512Kb of memory and a disk drive. You will undoubtedly find it useful for this and other programs to purchase a second disk drive or hard disk.

Users with only 512Kb of RAM may run out of memory when attempting to assemble larger programs or in other circumstances. The installation of a RAM-disk or other device on a 512Kb machine will restrict HiSoft Devpac.

If you are short of memory, remember that the least memory hungry thing is to assemble a one line program (consisting of an include statement) from the CLI. Upgrades to a megabyte of memory are available at very reasonable prices and we strongly recommend this, not just for HiSoft Devpac but for general use too.

Typography

In order to make the manual easy to read and to convey the maximum information as clearly as possible, we have adopted certain typefaces and typesstyles throughout the manual.

Typefaces

Times	General text.
Futura	Chapter and sub-Chapter headings and references to them.
Courier	Used to show something that is typed in at the keyboard or displayed on the screen. Predominantly used in program listings and references to function names, variables etc.
Avant Garde	Used for filenames, menu selections and button names. Also used to denote legends on single keys such as Alt (the Alternate key) and Ctrl (Control).

Typestyles

The *Italic* style is used mainly for *emphasis*.

Special Characters

[]	Within syntax descriptions, information enclosed in [] is optional.
...	Indicates repetition in syntax descriptions.
.	Vertically-spaced dots show that some part of a
.	program has been omitted.
.	

Acknowledgements

The trademarks (both registered and otherwise) of various companies are used throughout this manual. In particular:

HiSoft Devpac, *Power BASIC*, *HiSoft BASIC*, *GenAm* and *MonAm* are trademarks of HiSoft.

Amiga® is a registered trademark of Commodore Inc. *AmigaDOS*, *Kickstart*, *Intuition*, *Workbench* are trademarks of Commodore Inc.

We acknowledge any other trademark used but not listed above.

We would like to thank the following people for their invaluable help in the production of HiSoft Devpac and this manual:

Andy Pennell for his hard work in putting together the original Devpac, Julia for holding the fort when lesser people would have deserted, the lady with the dog (who was that girl?) and all the staff at The Old Bell!

A Quick Tutorial

This is deliberately a 'quick and dirty' tutorial so you can see how straightforward it is to edit, assemble and debug programs with Devpac Amiga.

In this tutorial we are going to assemble and run a simple program, which contains two errors and debug it. The program itself is intended to print a message.

To follow this tutorial you must already be in the editor. If you are not you should reboot your computer from the *backup* of Devpac Amiga disk 1 that is suitable for your version of Workbench; then open the disk icon and double-click on the Devpac icon.

After a short delay the screen will show an empty window; to load a file you should press the right mouse button, then move the mouse pointer over the **Project** menu and select **Open....** A file requester will appear; the file we want is in the examples drawer under the name of `demo.s`, so you should select this file and then click on **OK** or **Open**.

When the file is loaded the window will show the top lines of the file. If you want to have a quick look at the rest of the program you may press **Shift -↓** to move to the next page.

With most shorter programs it is best to have a trial assembly that doesn't produce a listing or binary file to check the syntax of the source and show up typing errors and so on. Move the mouse to

the **Program** menu and select **Check**.

The assembler will report an error, instruction not recognised, pressing any key will return you to the editor. The cursor will be placed on the incorrect line and the error message displayed in the window title bar.

The program line should be changed from `MOV.L` to `MOVE.L`, so do this, then select **Assemble** from the **Program** menu to assemble to memory. This is very much faster than assembling to disk and allows you to try things out immediately, which is exactly what we want.

The assembly worked this time.

If you are using an Amiga® with a 68000 processor (as opposed, to a 68020/68030/68040) don't run it yet there is a deliberate error and you will probably see a software error task held message from the system and you will have to reboot the machine! Otherwise you may run the program before proceeding.

The tool for finding bugs and checking programs is a debugger, so select **Debug** from the **Program** menu which will call the

debugger. This is described more fully later, but for now we just want to run the program from the debugger to 'catch' any problems and find out what causes them, so press **Ctrl-R** to run the program.

On a 68000 computer, the message **Address Error** will appear-at the bottom of the display, with the disassembly window showing the current instruction

```
start          MOVE.L      dosname,a1
```

This instruction causes an address error on a 68000 because the location `dosname` is at an odd address which cannot be accessed with the `MOVE.L` instruction. This is not the case on 68020s upwards and, if you are programming on such a machine, you should be aware of this since you may write code that works fine on your computer but crashes on a 68000.

To work on a 68000, there should a hash sign before `dosname` to put the *address* of `dosname` into the register `a1`. So, to return to the editor first press **Ctrl-Q** to ensure that the program will be killed and press **Y** to terminate the program and the debugger. We can now fix this bug in the source code.

Press **Ctrl-↑**, to go to the top of the file, then select **Find** from the **Search** menu. We are going to find the errant instruction so enter

```
move. 1
```

and press **Return** to start the search. Ahah! The first occurrence is the one we are looking for

```
start          move.l      dosname,a1
```

Add a hash to change it to

```
start          move.l      #dosname,a1
```

then assemble again. If you use **Run** from the **Program** menu you should see a message, press a key and the window will be closed, returning you to the editor.

Although the program now works we shall use MonAm, the debugger, to trace through the program, step by step. To do this select **Debug** from the **Program** menu, the debugger will appear with the message **Breakpoint**, showing your program.

There are various windows, the top one displaying the machine registers, the second a disassembly of the program, and the third some other memory.

If you look at window 2, the **Disassembly** window, you will see the current instruction, which in this case is

```
start      move.l #dosname, a1
```

As the debug option was specified in the source code all program symbols will appear in the debugger.

Let's look at the area around **dosname**. Press **A3** and you should see window 3's title inverted which means that it is selected. Next press **AA** and a requester will appear, asking **Window start address?** - to this enter

```
string
```

and press **Return**. This will re-display window 3 at that address, showing the message in both hex and ASCII.

Now to run the program; to execute the **move** instruction at the start of the program press **Ctrl-Z**. This will execute the current instruction and the screen will be updated to reflect the new values of the program counter and register **A1**. If you press **Ctrl-Z** again the next, **moveq**, instruction will be executed and **d0** will thus be modified. The following two instructions are the expansion of the

CALLEXEC OpenLibrary macro. Use **Ctrl-Z** to single step the move to **a6** instruction.

Next we have

```
JSR _LV0openLibrary(A6)
```

This is the call to the **exec** library. All calls to the Amiga's operating system have the same form. We don't want to single step this - we know that AmigaDOS works (at least most of the time!).

So, to treat this system call as one instruction, use **Ctrl-T**.

Now single step the next (**TST .L D0**). **D0** *should* be non-zero, so the **Z** flag won't be present in MonAm's register display area.

Now you can continue to press Ctrl-Z until you get to

```
JSR      _LV00Output(a6)
```

This is the call to the dos .library to find our output handle for displaying the message. Use Ctrl-T to skip over this. Use Ctrl-Z to continue single stepping until we get to

```
JSR      _LV0Write(a6)
```

This is the call that will actually write our string on the screen - let's make sure that the registers are set up correctly. d1 should have the output handle that came back from the _LV00Output call in d0. d2 should contain the address of the message.

Now use Ctrl-T to skip over the Write call. To check that it worked press the V key, there's the message; press any other key to return to MonAm (being careful not to activate any other windows meantime).

Now all that is left is our de-initialisation. You can use the Ctrl-Z and Ctrl-T commands to step through it as before.

The last instruction in the program is RTS. Single-stepping this will terminate MonAm, for now, and return to the editor.

That completes our quick tutorial.

Chapter 1

HiSoft Devpac 3

Introduction

Devpac 3 Disk Contents

Devpac Amiga is supplied on four 3.5" disks; the first set of two is intended for use under version 1.3 of the operating system whilst the second is for use with Release 2 to 3.1.

The first disk of each set is the *Program Disk* consisting of a Workbench disk and containing the Devpac drawer within which most of the tools can be found. You may boot your Amiga® from this disk and proceed to use Devpac immediately although you *must* ensure that you use the correct version for your machine.

The second disk of each set contains the full set of operating system 'include' files as supplied by Commodore. The majority of the Intuition, DOS, Graphics and Exec include files can be found on the program disk as a pre-assembled header file for compactness and speed.

Many users will find this sufficient for their requirements although more advanced users are urged to re-assemble this file as needed. Linkable libraries and Function Description files are also supplied on the *Include Disk*.

Please note that the following list of files is intended as a guide only and for actual disk contents you should refer to the Contents file. Subsequent versions of Devpac may contain extra files.

Program Disk

The Program Disk is a cut-down bootable Workbench disk containing the majority of development tools. Some directories and files of particular interest are:

Contents	The disk contents list for your version of HiSoft Devpac.
ReadMe	A text file including latest details about Devpac Amiga. Please read this file carefully before contacting our technical support department with any queries.
Devpac	This drawer contains the program tools detailed below.
Devpac/Examples	Some example programs including the short tutorial for this manual can be found in the Examples drawer.
Libs/arp.library	(1.3 version only) An additional library containing the File Requester used by the Devpac editor.
Prefs/Env-Archive/Devpac	The contents of the Env-Archive directory are automatically copied into ENV: at boot time by the Startup-Sequence script file. You may place preferences files or default icons within this directory (see the editor manual for details).
S/Startup-Sequence	The script file run automatically by AmigaDOS as part of the boot process. Advanced users may wish to modify this file in order to customise the disk.

The Devpac drawer

This drawer contains the programs and may be copied onto your hard drive as part of the installation procedure. Each of these programs is documented in detail elsewhere in this manual. Where disk space is limited, some tools may be found on the *Include Disk*.

Devpac	The multi-window editor and control program.
GenAm	The powerful Devpac 680x0 macro assembler.

system.gs	Pre-assembled system include file for use with the assembler.
MonAm	The debugger.
MonAm.libfile	A support file for the debugger enabling the automatic naming and display of many operating system library calls.
BLink	The AmigaDOS linker.
FD2LVO	A utility program to generate library offset include files from standard FD files.
SRSplit	A utility program for users of Motorola format S-Records which splits an S-Record file into its high and low byte components.

Include disk

All of the standard Commodore operating system include files can be found on this disk along with linkable libraries and FD files. Users may select which version of the disk they use according to the version of the operating system they are *developing for* rather than the appropriate version for their machine. The contents of the 1.3 and Release 2.0 versions may differ slightly.

include.i	A directory containing the system include files and LVO include files generated from the fd directory using the supplied FD2LVO program.
lib	Various Commodore supplied linkable libraries.
fd	The Function Description files giving function names, parameter names and register conventions for the system libraries and devices.
arp	(1.3 version only) Documentation for the ARP library containing a file requester and many useful support routines. Most of these capabilities are now provided as part of the Release 2 operating system.

Always make a back-up

Before using Devpac you should make a back-up copy of the distribution disk and put the original away in a safe place. It is not copy-protected to allow easy back-up and to avoid inconvenience. This disk may be backed-up using the Workbench or any back-up utility.

Before hiding away your master disk make a note in the box below of the serial number written on it. You will need to quote this if you require technical support.

Serial No.

Installation

If backing up your Devpac disks with Workbench, rename each one afterwards to remove the words 'copy of '. Please note that Program Disk 2.0 is a FastFileSystem disk and is unreadable from Workbench 1.3.

When installing on a hard disk, first copy the Devpac drawer by dragging its icon, then drag the appropriate Include drawer inside the new Devpac drawer.

You may wish to update the Include setting on the Assembly Control requester and the Default Tools of icons in Env-Archive (using Info from Workbench) to include the correct volume names for your system.

To install Devpac on a hard drive, we recommend that you drag the Devpac drawer icon onto your hard disk using the Workbench. Users of 1.3 will also wish to copy the `libs/arp.library` file to the `libs` directory of their boot partition. The `Env-Archive/Devpac` directory should also be copied to the hard drive for use with settings files and default icons.

We recommend copying the contents of one or both *Include Disks* (according to which operating system you are developing for) onto your hard disk for ease of access. The Devpac directory should normally be added to the AmigaDOS command search path to allow convenient use from the Shell, CLI or script files.

Floppy disk users are recommended to use the appropriate *Program Disk* as a basis for customisation, adding or removing further files as necessary. As with most Amiga® development packages, a second floppy disk drive is a considerable advantage, avoiding much disk swapping. CLI users may wish to convert the Startup-Sequence script file to create a new CLI window at boot time, automatically.

HiSoft Devpac functions to its best advantage with 1Mb of memory or more. However, it is perfectly possible to use the package on a 512Kb Amiga®; removing disk buffers, unwanted devices and programs will all release memory for use with Devpac.

Larger programs may be assembled by running the editor, assembler and debugger programs separately and through use of the assembler's low memory mode. Splitting source files into smaller sections which are included by a main file or assembled separately and later linked will also reduce memory usage.

Registration Card

Enclosed with this manual is a registration card which you should fill in and return to us. Without it you will not be entitled to your 30 day free technical support or upgrades within this period. Be sure to fill in all the details especially the serial number and version number.

The README File

As with all HiSoft products Devpac is continually being improved and the latest details that cannot be included in this manual may be found in the Readme file on the disk. This file should be read at this point, by double-clicking on its icon from the Workbench. It will also contain last-minute details on the installation process.

Chapter 2

HiSoft Devpac 3

Using the Editor

Introduction

HiSoft Devpac 3 (called simply HiSoft Devpac from now on) is a complete package for the production of fast, efficient assembly language programs on your Amiga® computer.

There is an *editor* for the creation and editing of your assembler source code, a *linker* for linking programs together, a *debugger* for helping you to stamp out those nasty bugs (problems) and, of course, an *assembler* to turn your source code into speedy, compact machine code.

We have also provided all of the Amiga® 'include' files (required when using the operating system) for both Workbench 1.3 and Release 2 together with a number of example programs.

This chapter looks at using the editor that is supplied as the heart of HiSoft Devpac and aims to give you a friendly introduction to most aspects of the package - it does not detail the assembler, linker or debugger or the technical aspects of the assembly process; these are covered in later chapters.

You may wish to take advantage of our special offers on a range of technical books that you will find invaluable if you are a serious assembler language programmer; for details, see the order form included with this package.

The Editor

The editor supplied with Hi Soft Devpac is fully integrated with the system which means that you can develop programs in an intuitive and interactive manner, creating and editing your programs in the same environment as running and debugging your finished masterpiece.

Moreover, those of you with strong preferences for your own editor can dispense with the HiSoft editor and use your own favourite package along with the stand-alone version of HiSoft Devpac, although you will lose the benefits of interactive development.

The editor for HiSoft Devpac is a multi-window screen editor which allows you to enter and edit text and save and load from disk. It also lets you print some or all of your text, search and replace text patterns and set bookmarks throughout the text so that you can find key points in your program quickly. In addition, you can define and use macros.

The editor is Intuition-based, which means it uses all the userfriendly features of Amiga® programs that you have become familiar with such as windows, menus and mice. However, if you're a die-hard used to the hostile world of computers before the advent of WIMPs, you'll be pleased to know you can do practically everything you'll want to do from the keyboard without having to touch a mouse.

The editor is 'RAM-based'; the file you are editing stays in memory for the whole time, so you don't have to wait while your disk grinds away loading different sections of the file as you edit. As all editing operations, including things like searching, are RAM-based they act extremely quickly. The file size is only limited by the amount of memory in your computer.

When you have typed in your programs it is not much use if you are unable to save them to disk, so the editor has a comprehensive range of save and load options, allowing you to save all or part of the text and to load other files into the middle of the current one, for example.

To get things to happen in the editor, there are various methods available to you. Features may be accessed in one or more of the following ways:

- Using a single key, such as a Function or cursor key;
- Clicking on a menu item, such as *Save*;
- Using a menu shortcut, by pressing the right *A* key in conjunction with another, such as *A F* for *Find*;
- Using the Control key (subsequently referred to as *Ctrl*) or Alternate key (called *Alt* from now on) in conjunction with another, such as *Alt-←* for *cursor word left*;
- Clicking on the screen, such as in a scroll bar.

The menu shortcuts have been chosen to be, hopefully, easy to remember and to be compatible with many other Amiga® programs.

Two versions of the Devpac editor are supplied. One is specifically designed to take advantage of Release 2 of the operating system and its many new features. There is also a 1.3 version which gives you the same look and feel on an Amiga® running earlier versions of the operating system. Be sure to use the correct version for your machine.

Starting the Editor

From the Workbench simply double-click on the Devpac icon to load it and display an empty window. If you have selected other icons as well as the editor icon (by Shift-clicking), these files will be opened as projects for editing when you run the editor.

To run the editor from the CLI, type its name (Devpac) followed by an optional list of file pathnames - a file extension of *.s* will be appended to filenames where necessary.

To find its startup settings, the editor searches for a file called Devpac.prefs, firstly in the current directory then in the editor's directory and finally in ENV:Devpac - if it cannot find this file it will use its built-in defaults. You may override this search from the Workbench by adding a *SETTINGS=<pref_name> tooltype* to a project icon or to the Devpac icon.

Default icons may be provided by placing them in the ENV:Devpac directory. An icon named `def s` will be used as the default for all files ending in `s` - if no such icon is found for a particular file type the `def_project` icon will be used instead.

We will return to a full description of projects, windows etc. after some general information about Devpac's use of gadgets and the like.

Requesters and gadgets

The editor makes extensive use of requesters which employ the Workbench 2 user interface (even if you are running under AmigaDOS 1.3) so it is worth recalling how to use these requesters and their associated gadgets. Note that most Devpac requesters do not restrict you from further editing.

If you require more detail than is given below, you should look in the *Amiga User Interface Style Guide* or the *Using the System Software* user guide supplied with your Amiga®.

Keyboard shortcuts

Before we describe the different types of gadgets available, we should mention that most of them can be accessed either using the mouse or the keyboard. Keyboard shortcuts may be used whenever a text gadget is *not* active and the key that addresses the gadget is shown underlined in the descriptive text next to the particular gadget. For example, in the Find requester shown below, you can use N or n to find the next occurrence of the defined string or C (or c) to cancel the operation.

As we have said, the editor's requesters contain various gadgets of differing types and we shall now describe them.

Action gadgets or buttons

Action gadgets are boxes containing a description of the action that will take place should you left-click on the box or button.



action gadgets or buttons

To activate the action displayed by the button, point at it and left-click - the action will be triggered when you let go of the mouse button; this allows you to move off the gadget while still keeping the mouse button depressed if you change your mind.

Sometimes, clicking on a button will cause another requester or window to be displayed. If this is the case, the action gadget's text will end in an ellipsis (three periods):



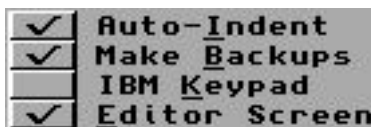
the Set... buttons lead to a file requester

The Cancel button will always be positioned at the bottom right of the requester with a keyboard shortcut of C and will abort the requester, leaving everything as it was before the requester was activated.

Many requesters also have a Use button which accepts any changes you have made and closes the requester. Clicking on a requester's close window gadget or pressing Esc has the same effect.

Check box gadgets

A check box is a small square that is either empty or contains a check mark or tick. This type of gadget allows you to turn some action on (checked) or off (blank) - simply click on it, with the left mouse button, to change its state.



check box gadgets with keyboard shortcuts

Cycle gadgets

Cycle gadgets allow you to choose one option from a list of several related items. To change to another option you must left-click anywhere in the gadget which will cycle forward through the list. If you hold the Shift key down while left-clicking, the list will cycle backwards. Here's an example:



cycle gadgets allow selection from a list

As usual, you can use any keyboard shortcut instead of the mouse - again use Shift and the shortcut to go backwards through the list.

When you reach the end of the list, it will wrap round to the first item.

Radio buttons

Radio buttons are also used when you have to make a choice of one item from a short list of alternatives.



Radio buttons used for a list

In the above example, you can choose to print the whole file, a marked block or a block defined by a line number range. When you choose one, the previous option becomes de-selected. Use Shift and the keyboard shortcut to move back through the list.

Text gadgets

Text gadgets allow you to enter strings of characters or numeric values.



text gadgets

To type into a text gadget you must activate it to obtain a cursor. Simply left-click inside the text gadget or press its keyboard shortcut to activate the gadget. Some commonly used requesters automatically activate a text gadget so that you can type directly into it as soon as the requester pops up.

As you are entering text, certain keys allow editing of the string:

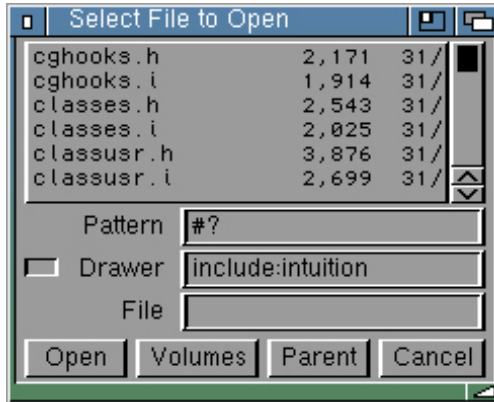
←, →	move the cursor left or right through the text
Shift-←, Shift-→	move the cursor to the start of the line or to the end of the line
Backspace	delete the character behind the cursor
Shift-Backspace, Shift-Del	(release 2 only) delete from the cursor to the start or end of line.
Del	delete the character under the cursor
A X	delete the entire line
A Q	restore the text to its former state
Tab, Shift-Tab	(release 2 only) activate the next or previous text gadget
Return	exit the text gadget

key commands available within text gadgets

To exit a text gadget simply press Return or left-click somewhere else in the requester.

File requesters

Whenever a file or directory name is needed, the Devpac editor will produce a File Requester which allows you to locate the desired object and select it. The Workbench 2 version uses the standard ASL file requester whilst the 1.3 version utilises the ARP requester. The operation of each File Requester is similar and straightforward. However, a brief summary is given below.



the Workbench file requester

The requester title bar shows the type of object which you are required to select and the operation that will be performed on it. Save requesters appear in a different colour. By using the scroll gadgets you may move about the list of files and directories; clicking on a name will enter it into the appropriate text gadget. You may also choose to type into these gadgets to select a known file quickly.

The four action gadgets at the base of the requester allow you to proceed with the operation (this button will be labelled accordingly), show all available volume names and AmigaDOS assignments, move into the parent of the current drawer and cancel the operation (the window close gadget is also available for this purpose). Double-clicking over a name will both select the file and proceed, which normally passes the filename back to the calling program.

If the object you select does not exist or is of the wrong type you will be given a chance to reselect. Any errors are reported in another requester.

The ASL file requester also has the added benefit of a sizing gadget (the requester size and position will be remembered when you Save Settings) and a pull down menu containing useful Amiga® key shortcuts. When you are saving a file, typing a drawer name which does not exist gives you the chance to create that drawer.



the simple file requester

There is also a simple file requester which will appear in a number of situations. Firstly, if the editor is unable to open the ARP or ASL library (according to the version) from your LIBS: drawer, the simple requester will be used instead. It will also be used if there is insufficient free memory to open the full file requester. Note that, in extreme low memory situations, it may not be possible to open a requester at all, in which case you may need to close some unmodified projects or quit any other applications which are running.

Finally, you may obtain the simple file requester by holding down the Shift modifier when selecting a filing command from the menu or with the keyboard. For example, Shift-A A will allow Save As... from the simple file requester. This can be useful if some other disk-intensive program is working and you do not wish to affect its performance.

Confirmation requesters

One other type of requester which you will see appears when it is necessary for you to make a choice before proceeding. Perhaps the most common occurrence of this is the modified protect requester:



This is produced if you select some action which would lose any changes made to a project. By clicking on the gadgets or pressing the first letter of their titles you can choose to save the project, to abandon the changes and go ahead with the action or to back out, leaving everything as it was before you selected the action.

Another way of choosing an action is to hold down the left-Amiga key and press either V for the leftmost button or B for the rightmost button which is always Cancel. Remember that the continue action is placed at the left and Cancel at the right. The middle button often causes an action which would lose or overwrite some data.

Note that, unlike most Devpac requesters, these will block any further input to the editor until you have chosen an action.

Menus and sub-menus

Menus allow you to access editor commands quickly and easily while giving you the opportunity to browse through the various choices. To access the Devpac menus simply press the right mouse button and hold it down.

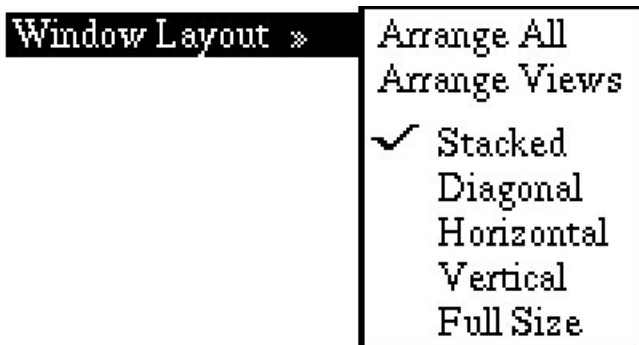


a typical Devpac menu

To select from the menu, still holding the right button down, move ' the pointer over one of the menu titles which appear at the top of the screen. Once the menu pops up, drag the selection bar to the required choice and let go of the mouse button. Many menu commands have keyboard shortcuts accessed by holding the right Amiga key down and pressing the relevant command key as shown on the menu.

To select a sequence of menu items while still viewing the menu, keep your finger on the right mouse button and click on the left mouse button when you are over the relevant choice - you will then still be able to move up and down the menu and perhaps select another item.

Some options are accessed via 'sub-menus' which are shown by the >> symbol to the right of the menu item; for example, if you move the mouse over the Window Layout selection on the Window menu, a sub-menu like this will appear:

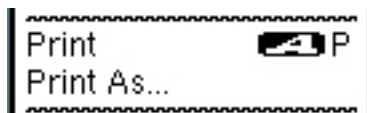


A sub-menu

You can then move the mouse to the right to select the particular item that you want. To cancel the operation just let go of the mouse without selecting a sub-item or move to another item on the main menu.

Some menu items can have a 'tick' or check mark next to them. Selecting such an item will select or de-select that choice. Once again you may use left-clicking to select a number of these options. The Window Layout sub-menu uses such items to give you a choice between the various layout styles.

If a menu item leads to a requester containing further choices, the text of the menu item will be followed by an ellipsis (three dots) as shown below:

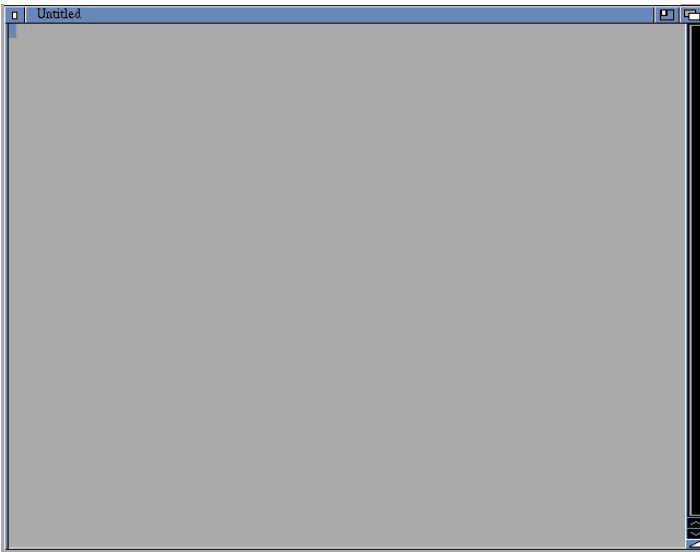


Print As... leads to a requester

The Editor's windows

Having loaded HiSoft Devpac, you will be presented with an empty window and a graphic block, which is the *cursor*, in the top left-hand corner.

The window used by the editor is a standard Intuition window, so you can move it around by using the *Title bar* on the top of it, you can change its size by dragging on the *Sizing gadget*, close it with the *Close gadget*, bring it to the front or send it to the back with the *Depth gadget(s)* and make it full size (and back again) by clicking on the *Zoom gadget* (AmigaDOS 2 and greater). You can also move the view on the text by using the *Scroll gadgets*.



the HiSoft Devpac window under AmigaDOS 2

Windows and Projects

We make the distinction between *windows* and *projects*: a project is a file with any number of windows open on it. You can also edit many different projects at any one time, each of which may have many windows.

To start a new project select New from the Project menu or type **A N** - this will give you an empty, untitled window. To open an existing project from disk you should use the Open command (**A O**) on the Project menu which will bring up a file requester allowing you to select a file to edit.



the Project menu

Once you have selected a file to edit you can open another window on that file by using **A-W** (New Window on the Window menu); subsequently close a window by left-clicking on its close gadget or by typing **Shift-A W**

To finish a project (and close all of its windows), select Close from the Project menu, closing the last window on a project has the same effect. The other selections on the Project menu will be discussed below under the appropriate heading.

Switching Windows

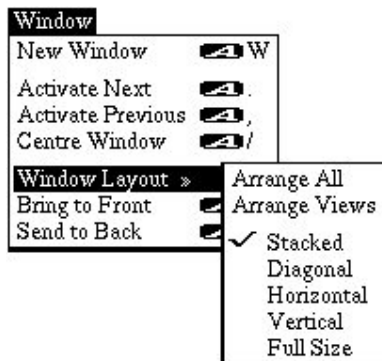
The editor has support for any number of windows, memory permitting, you can even have many windows (views) open on one file. Once you have some windows open, you can move between them in a variety of ways:

Firstly, you can select a window (if you can see it) by clicking on it with the mouse but this will not bring it to the front - you must use the window depth gadgets or **Shift-A >** to achieve that.

Under AmigaDOS 1.3 there are two window depth gadgets; one of these sends a window to the back and the other brings it to the front. We have implemented two keyboard shortcuts for these actions – $\mathcal{A} >$ (Shift- $\mathcal{A}$.) for bring to front and $\mathcal{A} <$ (Shift- $\mathcal{A}$,) for send to back. With AmigaDOS 2 there is only one depth gadget which you can Shift-click to send the window to the back. These commands leave the window upon which the action was performed selected, so that, if you send a window to the back revealing another window, this new window will not be selected (and therefore you will not be able to type into it) until you click on it.

A useful action is to cycle through all the available editor windows using Activate Next and Activate Previous commands from the Window menu (the keyboard shortcuts are $\mathcal{A} .$ and $\mathcal{A} ,$, respectively); these commands bring the relevant window to the front and select it, ready for use.

The windows can be organised in a number of ways and you can select this using Window Layout on the Window menu.



the Window Layout sub-menu

First of all, choose which layout you would like;

- | | |
|----------|--|
| Stacked | re-sizes the windows to the same width and arranges them so that you can see the title bar of each window (depending on which window is at the front). |
| Diagonal | arranges all the windows diagonally and changes their size so that you can see the top right area of all the windows (and hence their depth gadgets). |

Horizontal	organises the windows one above the other and resizes them so that you can see the whole width of every window - this is most useful for a small number of windows.
Vertical	is like Horizontal except that the windows are arranged side by side.
Full Size	means that each window is made the size of whole screen and placed one behind the other.

Having chosen the desired arrangement (this will not actually change any windows) you can then Arrange All windows that are open or only Arrange Views - this means only arrange the windows associated with the current project.

Try this out for yourself to get the idea of how the different arrangements work. The sub-menu works best if you left-click over a layout style then release the right mouse button over one of the Arrange items.

Entering text and moving the cursor

To enter text, simply type on the keyboard and at the end of each line press the Return key (or the Enter key on the numeric pad) to start the next line. You can correct your mistakes by pressing the Backspace key, which deletes the character to the left of the cursor, or the Delete key, which removes the character under the cursor.

Undo Line

The Devpac editor keeps a copy of the line that you are working on so that as long as you have not moved off the current line, you can undo the changes you have made and recover the line as it was before you started editing it.

To restore the original line select Undo Line from the Edit menu or press **Alt Z**. Remember, this will always work as long as the cursor is still within the line that you wish to recover.

Cursor keys and the mouse

You use the cursor keys, together with various *keyboard modifiers*, to move quickly around the file without needing to touch the mouse. The modifiers used are Shift, Ctrl and Alt and their use conforms to the guidelines given in the *Amiga User Interface Style Guide* which are as follows:

Using the cursor keys by themselves will move the text cursor around the file by a character (← →) or a line (↑ ↓) at a time, scrolling the display where necessary.

Holding the Shift key down while using the cursor keys will move to the edge of the window (i.e, the top/bottom/left/right) or show a new windowful of text if the cursor is already at the extreme of the window. Shift-← or Shift-→ have the effect of moving to the beginning or end of the current line. Shift also modifies the Backspace and Del keys in a similar way.

Using the Ctrl key with the cursor keys moves to the appropriate extreme of the *project* i.e. the beginning or end of line (Ctrl-← or Ctrl-→) and the top or bottom of the file (Ctrl-↑ and Ctrl-↓).

Finally the Alt modifier works on words so that Alt-← or Alt-→ move the cursor a word left and a word right respectively whilst Alt-↑ and Alt-↓ move on a page basis (like Shift). When used with Backspace or Del, Alt has the effect of deleting a word at a time.

The function of the Shift and Alt modifiers can be swapped (see Settings); this maintains some compatibility with previous Devpac editors. It is also possible to set up the numeric keypad to act like an IBM PC keypad so that you can perform many cursor operations using its keys - see the Settings section for details.

You can use the mouse to move the cursor to a specific position on the screen by pointing and clicking. As usual, you can use the slider gadget to scroll through the file, in which case the cursor will move with the text. You can scroll the window up and down on a line basis by clicking on the vertical scroll gadgets.

If you position the cursor past the right-hand end of the line and type some text at that point the editor will automatically add the text to the real end of the line. If you type in long lines the window display will scroll sideways if required.

All the above commands along with the other keyboard shortcuts are summarised at the end of this chapter.

Tab key

The Tab key inserts a special character (ASCII code 9) into the text, which on the screen looks like a number of spaces, but is rather different. Pressing Tab aligns the cursor onto the next 'multiple of 8' column (by default), so if you press it at the start of a line (column 1) the cursor moves to the next multiple of 8, +1, which is column 9.

Tabs are very useful indeed for making items line up vertically and their main use in HiSoft Devpac is for such things as indenting structured program lines. When you delete a tab the line closes up as if a number of spaces had been removed. The advantage of tabs is that they take up only 1 byte of memory, but can show on screen as many more.

You can change the tab size for each individual file - see Settings for details.

Return key

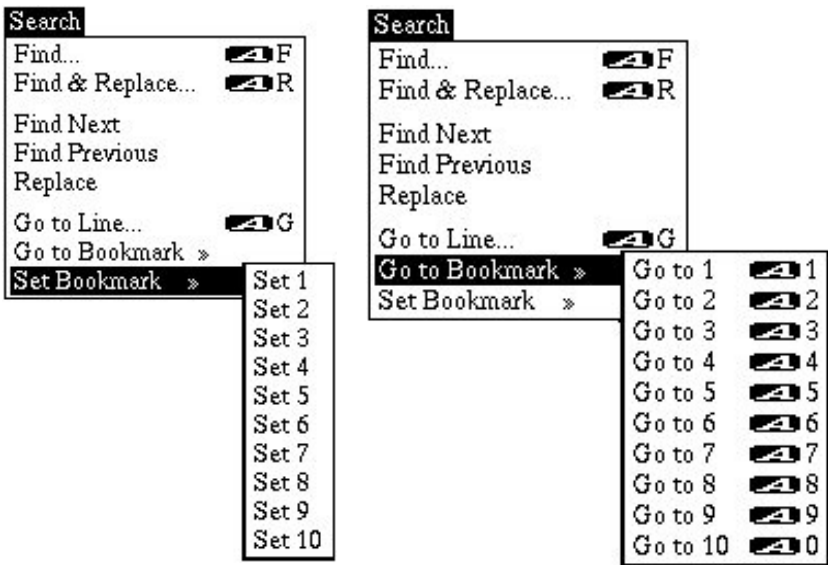
In addition to entering a line of text, the return key can be used to insert a blank line into the text by pressing Ctrl-Return.

When Return is pressed in the centre of a line, the line will be split into two. Ctrl-= can also be used to split a line at the cursor position whilst Ctrl-- (hyphen) will join the next line to the end of the current one.

Bookmarks

A further way to navigate your text is through the use of bookmarks. A bookmark is set by selecting the appropriate item on the Set Bookmark sub-menu from the Search menu or by using Shift-*A* and one of the digit keys on the top row of the keyboard, zero for 10 (e.g. Shift – *A* 4 sets bookmark 4).

Having set a bookmark, you are then able to select the corresponding item on the Goto Bookmark sub-menu to return the cursor to the line on which you set the bookmark; or use *A* and the appropriate digit e.g. *A* 2.



A possible total of 10 bookmarks are allowed per project which are saved along with the project only if Create Icons? is selected. Bookmarks are lost if the line they are on is deleted.

Deleting text

Backspace key

The Backspace key (to the left of the Del key) removes text to the left of the cursor. When pressed on its own it will delete a single character but when used in conjunction with the Alt or Shift modifier keys, it will delete the previous word or to the start of the the line.

If you backspace off the very beginning of a line this will normally remove the 'invisible' return character and join the line to the end of the previous line (although you can change this behaviour using Set Settings...). Backspacing when the cursor is past the end of the line will move the cursor to the end of the line without deleting a character.

Del key

The Del key removes text to the right of the cursor. Similar to Backspace, just pressing Del will delete the character under the cursor whilst Alt and Shift delete by word or to the end of the line (also available as Ctrl-Q for compatibility).

Del will join lines together if used at the end of a line, unless you have changed this behaviour using Set Settings.... Using Del when the cursor is past the end of the line will simply move the cursor to the end of the line.

Delete line

The current line can be deleted from the text by pressing  Backspace, Ctrl-Backspace or Ctrl-Y. The deleted line is remembered and may be later inserted back into the text.

Undelete Line

When a line is deleted using the above command it is preserved in an internal buffer, and can be re-inserted into the text by selecting Undelete Line from the edit menu or pressing Ctrl-U. This can be done as many times as required which is particularly useful for repeating similar lines or swapping individual lines over.

Delete block

A marked block may be deleted from the text by pressing **⌘ Del** or **Ctrl-Del** or **Shift-F3**. These are all equivalent to **Erase** on the **Edit** menu.

The following table summarises the behaviour of **Backspace** and **Del** when used with various modifier keys:

	Alt	Shift	Ctrl or Amiga
Backspace	Delete previous word	Delete to start of line	Delete line (also Ctrl-Y)
Del	Delete next word	Delete to end of line (also Ctrl-Q)	Delete block (also Shift-F5)

Remember that the current line is buffered, so you can undo all changes made to the line by using **⌘ Z Undo Line**.

Block Commands

A *block* is a marked section of text which may be copied to another section, deleted, printed or saved onto disk. The editor will report **Block required** if a marked block is needed. Blocks may be marked using the mouse or the **Mark Block (A-B)** command on the **Edit** menu. After starting a block mark, moving the cursor will set the current block, until the **Mark Block** command is used again.

In order to maintain backward compatibility, we have retained many of the block keyboard shortcuts used in earlier versions of **Devpac**, as you will see as you read on.

Marking a block

The simplest way to mark a block is to left-click on the first character in the block and drag the mouse to the end of the block. The block is highlighted by showing the text in reverse as you drag the mouse. When you move the mouse past the edge of the window, the window will automatically scroll to allow blocks larger than the window size to be marked. You may mark a block by clicking at the end and dragging back if you wish.

Double-clicking will cause the word under the mouse to be marked as the block. If you double-click and then drag, text will be highlighted a word at a time.

A second way of marking a block is to place the cursor at one end of the block, either with the movement keys or by pointing and clicking. Then, holding down the Shift key, click at the other end of the block. This technique is often useful for marking a very large block as it can be used to extend an existing block.

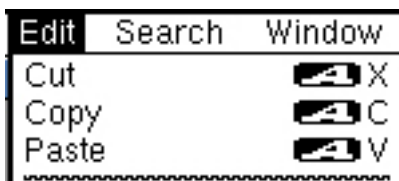
The start of a block may also be marked by moving the cursor to the required place and pressing F1. The end of a block can be marked by moving the cursor and pressing key F2. The start and end of a block do not have to be marked in a specific order - if it is more convenient, you may mark the end of the block first.

To unmark any selected block, press Esc. You may also go to the beginning or end of the currently marked block by pressing Shift-F1 or Shift-F2.

The Clipboard: Cut, Copy & Paste

HiSoft Devpac supports the Amiga® clipboard, allowing you to not only cut and paste text within Devpac, but to share data between many different applications. The clipboard holds the most recently cut or copied block of text ready for subsequent pasting to another location.

Once you have marked a block you may copy it into the clipboard using Copy from the Edit menu. The text will remain in the file and the copy of the text held in the clipboard may then be inserted at another position by moving the cursor there and selecting Paste.



cut, copy and paste

The current block may be removed and placed in the clipboard using Cut from the Edit menu; selecting Paste will then insert the block that was cut as before. To move a block of text involves marking the text, selecting Cut, moving to a new position and then Paste.

Cut block

A marked block may be deleted from the text by selecting Cut on the Edit menu or by pressing **A X** or Shift-F5. A cut block is remembered in the clipboard for later use.

Copy block

The current marked block may be copied to the clipboard using Copy on the Edit menu or by pressing **A C** or Shift-F4. The block is automatically unmarked to show that it has been copied into the clipboard successfully.

This command can be very useful for moving blocks of text between different files by loading the first, marking a block, copying it to the clipboard then switching to another window or loading the other file and pasting the clipboard into it.

Paste block

The contents of the clipboard may be pasted at the cursor position by selecting Paste from the Edit menu or by pressing **A V** or F5. If the clipboard is empty or contains information other than text, the message No text in clipboard will be reported. If the editor is unable to access the clipboard, a Clipboard error is given.

Delete block

If you wish to delete a block without placing it in the clipboard, choose Erase on the Edit menu or press **A Del** or Ctrl-Del or Shift-F3. There is no way of retrieving text which is deleted in this way.

Duplicate block

A marked block may be copied, memory permitting, to another part of the text by moving the cursor to where you want the block copied and pressing key F4. If you try to copy a block into a part of itself, the message Cannot copy inside block will appear. This command does not use the clipboard and is supplied for compatibility with previous versions of Devpac.

Save block

Once a block has been marked, it can be saved to disk by selecting Save Block... from the Edit menu or by pressing key F3. Assuming a valid block has been marked, a file requester will appear, allowing you to select a suitable drive, drawer and filename in which to save the block. You will be warned if you specify a filename which already exists in which case a backup will be made if requested in *Settings*.

To insert any previously saved file at the cursor position you may select Paste File on the Edit menu.

Printing a block

A marked block may be sent to the printer or a file by selecting Print As... on the Project menu. A requester will appear and you should choose Selected Block, the number of copies and the output name.

See the *Printing* section for more detail on the use of this requester.

Searching



The commands on the Search menu may be used for finding and perhaps replacing existing text. This is done by first selecting Find... (A F) or Find & Replace... (A R) to specify the search and replace text and then using either the requester or the Find Next, Find Previous and Replace commands.

The Find requester is a subset of the Find and Replace requester described below with facilities for searching only.



the Find & Replace requester

In the example above options has been entered as the find text and configuration as the text to replace it with.

If you click on Cancel, the requester will disappear, taking no action and restoring the previous find and replace text. If you click Find Next (or press Return) the search will start forwards, while clicking on Previous will start the search backwards.

If the search is successful, the window will be re-drawn with the cursor positioned at the start of the string. If the string could not be found, the message Not found will appear in the window title bar and the cursor will remain unmoved.

You may continue the search with Find Next and Find Previous or by activating an editor window and selecting Find Next or Find Previous from the Search menu. The keyboard shortcuts for these commands are Ctrl-N and Ctrl-P. Note that you do not have to close the requester in order to continue editing although this can be done via the close gadget or by pressing Esc.

Whether test is treated as the same as TEST or Test etc. depends on whether the Case Sensitive check box is selected. Normally, searching is case insensitive, meaning that upper and lower case letters are treated the same. However, with Case Sensitive, an exact match for the find text must be found before searching will stop.

Optionally, you may search for Whole Words by selecting the second check box. This will prevent a search for the word in, for example, matching the word finish. It will still find the word in *first-in-first-out* however because the hyphen has been used to separate the words.

Replacing Text

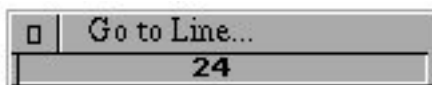
Having found an occurrence of the find text, it can be changed to the replace text by clicking on the Replace button. This is identical to selecting Replace from the Edit menu or pressing Ctrl-R whilst editing. Having replaced it, the editor will then search for further occurrences.

You may of course replace several occurrences of the text by selecting Replace several times in succession however, if you wish to replace every occurrence of the find string with the replace string starting from the cursor position, click on the Replace All gadget. This may take some time if there are a large number of occurrences. There is deliberately no alternative way of selecting Replace All in order to prevent it from being chosen accidentally.

You may search and replace Tab characters, line feed characters or any other control characters in a similar way by simply entering them in the appropriate text gadget. This facility can be used to search for something at the beginning or end of a line by including a line feed character (Ctrl-J) at the start or end of the find text. Note that AmigaDOS 2 users may have to hold down the left Amiga key while typing these or other control characters.

Go to line

To move the cursor to a specific line in the text, select Go to Line... from the Search menu, or press *A.G.* A small requester will appear, allowing you to enter the line number and press Return.



The cursor will move to the specified line, centring the window over it. If the line does not exist, you will be positioned over the start or end of file. Pressing Return with no line number or clicking the close gadget will abort the operation.

Another fast way of moving around the file is by dragging the window scroll bar, which works in the usual fashion.

Bookmarks

The Go to Bookmark sub-menu may be used to move to one of the project's 10 bookmarks previously set up via the Set Bookmark commands. Further details can be found in the Bookmarks section.

Disk Operations



The Project menu contains many operations that involve disk files; you can save and load your source file, insert text into your source, print a file and more.

New

Select New to open an untitled project and window ready for entering text. The number of projects is limited by available memory only.

Opening files

To load in a text file, select Open... from the Project menu, or press `A O`. This will open a new window if the current one is in use and then a file selector will appear, allowing you to specify the volume, drawer and filename. Assuming you do not Cancel, the editor will attempt to load the file. If it will fit, the file is loaded into memory and the window is re-drawn.

If an error occurs, a requester appears showing the error and filename, giving you a chance to Retry or Cancel as with all file operations. Some less common errors are reported as AmigaDOS error numbers as documented in your system manual.

A number of files may be opened automatically when HiSoft Devpac is started by selecting project icons from the Workbench and Shift-double-clicking the editor icon, or by specifying a number of filenames on the command line.

Close

Closes the current project and all of its windows. This will also happen when you close the last window on a project either using the close gadget or by pressing Shift ⌘ W. If the file being edited has changed since it was loaded or is a new file, you will be given a chance to save it before the window is closed.



Closing the last project will present you with the above requester which allows you to quit Devpac or open a new project.

Save

To save the text you are editing, select Save from the Project menu or press ⌘ S. If the file was loaded from disk, it will be backed up (if requested in *Settings*) and the new version is written to disk. Untitled projects will produce a file requester, allowing you to select a name for the project like Save As...

If Create Icons? on the Settings menu is selected, the editor will attempt to generate a suitable Workbench icon for the file.

Devpac also saves the current tab size and positions of any bookmarks for the project in its icon. This information can only be saved to disk (and subsequently re-loaded) when Create Icons? is enabled.

Save As...

Save As... on the Project menu or ⌘ A allows you to save a project in a different place or under a new name. It will be selected automatically if you Save an untitled project.

The File Selector will appear, giving the name and drawer where the file is currently located (if any). You may select the drawer and filename in the normal way. Clicking Save or pressing Return will then save the file onto disk. If you click on Cancel the text will not be saved.

If the filename you select already exists, a requester will appear asking you if you wish to overwrite this file, select a new name or cancel the operation. To save the file under the same name, use the Save command.

Backups and icons are also created if selected as described under *Settings*.

Save Changes

If you are editing a number of files and wish to save all of them to disk, Save Changes is equivalent to selecting Save for each modified file. You may use this before quitting the editor or for safety before running a program which you are testing.

Last Saved

The shortcut Shift-*A*-L may be used to activate the 'Last Saved' command. This command will load the last saved version of the file on disk. Somewhat similar to an 'undo' command. 'Last Saved' is greyed-out on a project that has not been saved previously.

Inserting a file

To read a file from disk and insert it into a project, select Insert File from the Edit menu. The File Selector will appear allowing you to select a filename or to cancel. The file will be read from disk and inserted, memory permitting, at the current cursor position.

Deleting files

This command is only available in the 1.3 version. You may want to delete a file from disk (if for instance you have run out of disk space whilst trying to save); click on Delete File. The File Selector will appear, allowing you to select a suitable disk and filename. Clicking OK or pressing Return will delete the file and its icon from disk and prompt for another one. Click on Cancel to stop (this will not cause any files to be deleted).

If an error occurs, a requester will appear showing an AmigaDOS error, the exact meaning of which can be found in your Amiga® manuals. Note that it is only possible to delete a drawer after any files which it contains have been deleted.

Quitting HiSoft Devpac

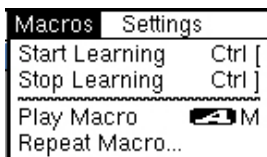
To leave HiSoft Devpac, select Quit Devpac from the Project menu or press. **⌘ Q**. If any changes have been made to the text and not saved to disk, a modified project requester will appear for each changed file giving you a chance to save or abandon the file. Selecting Cancel will abort the quit and resume editing.

If you have a large number of modified projects, all of which you wish to save to disk, it can be quicker to select Save Changes before quitting.

Macros

Macros provide a simple way of teaching the Devpac editor to perform a sequence of actions. This facility can be used for simple operations, such as indenting a number of lines by a single tab, or more complex ones such as adding a comment to the end of all lines containing the instruction PMOVE.

The Macro menu gives access to the four commands used to record and replay macros.



the macro menu

To begin a macro you should select Start Learning from the Macro menu or press Ctrl-]. The editor will subsequently remember each action you perform including all of the menu items, cursor movement commands and typed text. Mouse or requester operations are not remembered, only editor commands will be recorded.

Once you have completed your chosen sequence of actions, select Stop Learning or press Ctrl-] . The macro can now be played back to repeat the recorded sequence.

Two playback commands are available. Selecting Play Macro or ⌘ M will replay the macro once whilst Repeat Macro... or Shift ⌘ M will play the macro several times in succession.

If you choose to repeat a macro, a number requester similar to the Go to Line requester will appear allowing you to enter the number of iterations and press Return. Clicking the close gadget or pressing Return without entering a number will cancel the operation.

The following sequence of actions would be required in order to record a macro which indented a number of lines by a single tab:

- select Start Learning (Ctrl-]).
- use Ctrl-← to move to the beginning of the line, press Tab and then cursor down a line.
- select Stop Learning (Ctrl-]) to complete the macro which will consist of these three actions.
- choose Repeat Macro and enter the number 10 to proceed to indent the next ten lines in exactly the same way.

Settings

Selecting Settings... from the Settings menu will produce a requester like this:



The editor preferences

This allows you to configure the editor as you like to use it; you can then save your customisation to disk so that the editor will always behave the same way. Selecting Use will accept any changes you have made without saving them to disk. Changes will be lost at the end of the current session unless subsequently saved.

Here are the different settings that you can change.

Tab Size

By default, the tab setting is 8, but this may be changed to any value from 1 to 20. This value will be used for the current project and as the default for any new projects. Tab sizes are saved for each individual project if Create Icons? is selected.

End of Line

There are three possible choices of the action that is taken when the editor reaches the end of a line. The default behaviour allows the most freedom although you can also choose for the cursor to wrap to the next line if you step past the end of a line or to treat the start and end line as terminators. This latter behaviour allows you to hold down the Del key to delete to the end of the line without it proceeding to delete the rest of your text.

The best way to find out which you prefer is to try using each setting.

Word qualifier key

By default, the Alt modifier is taken to mean 'by word' and Shift is 'by line' as is standard for the Amiga®. However, users whose fingers are accustomed to other systems can swap this behaviour via the Word Key setting.

Auto-Indent lines

Selecting this option gives automatic line indenting. When active, an indent is added to the start of each new line created when you press Return or split a line.

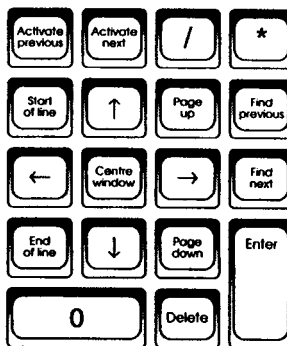
The contents of the indent of the new line is taken from the white space (i.e. tabs and/or spaces) at the start of the previous line. This allows you to lay out your program neatly by simply pressing Return.

Make Backups

Selecting this option causes the editor to make a backup (by adding the extension .BAK) when saving text files.

IBM Keypad

This option enables the use of the numeric keypad in an IBM PClike way by default allowing single key presses for cursor functions. The keypad works as shown in the diagram below:



When this option is not selected, the keyboard reverts to returning the digits etc. although these functions may still be accessed by using the Ctrl modifier in conjunction with the numeric pad keys.

At all times, the Shift modifier and the numeric pad cursor keys will scroll the window in the appropriate direction whilst keeping the cursor in the same position over the text.

Editor Screen

The Devpac editor may be run either from the Workbench screen or its own screen if preferred. Selecting Editor Screen and saving the settings will cause a new screen to be opened when the editor starts up.

The Workbench 2 version opens on a public screen named DEVPAC.1 by default. You are free to use this screen for other programs or for opening Shells via the SCREEN keyword of the CON: handler.

Create Icons

With Create Icons? on the Settings menu selected, the editor will attempt to generate a suitable Workbench project icon for any saved file. The icon image is taken from the ENV:Devpac or ENV:Sys directory and will match the file extension where possible. For example, the icons `def_s` and `def_i` would be used for the files `amiga.s` and `menu.i` respectively. If these could not be found then the icon `def_project` would be used instead.

You may create these icons with the Workbench icon editor and include a suitable default tool (this can be used to automatically run the editor when you double-click the icon) and other tooltypes as desired. Some common icon names are `def.s` and `def.i` for source files with `def_project` for other files, `def_prefs` for settings files, `def_opts` for assembler options files and `def_tool` for assembled programs.

Saving settings

To save the settings file you can choose Save Settings or Save Settings As... from the Settings menu. This latter command saves all editor preferences to the current settings file or creates a default file if no preferences have been loaded. Selecting the Save gadget on the Settings requester has the same effect. An icon of type `def_prefs` will also be created if you have selected Create Icons?.

In addition to saving the editor configuration, the current font and find and print settings are also saved. The current window position and size are saved as the defaults for new projects and the relative position of all requesters are remembered.

The settings file itself is formatted in a similar way to Workbench icon tooltypes as a list of keywords in upper case followed by an equals sign and their associated setting. Advanced users may wish to modify the files textually within the editor in order to override only certain defaults or to specify an exact window position etc.

Multiple settings files

The usual name for settings files is `Devpac.prefs`. If you want to call your settings file a different name, you can use Save Settings As... which will allow selection of a name and drawer for the file via the File Requester.

When the editor is loaded, it looks for the `Devpac.prefs` configuration file firstly in the current directory (which is the project icon drawer when started from Workbench by doubleclicking), then in the editor directory (used by Save Settings when no file is available) and finally in `ENV:Devpac` (for the convenience of network users). This can be overridden by the use of the `SETTINGS=<file>` tooltype which can be added to editor or project icons via the Workbench Info or Information option.

You can take advantage of this if you intend working on several projects, each requiring different settings, by saving settings files into each project drawer and a default settings file in the editor or `ENV:Devpac` directory.

Loading settings

You may use Load Settings... from the Settings menu to select and load editor settings files from the File Requester. Workbench 2 users may wish to keep a number of settings files in the SYS:Prefs/Presets drawer.

Assembler settings

The Assembler sub-menu on the Settings menu allows you to select the options used by GenAm when assembling programs. Although full details of each option can be found in the assembler chapter of this manual, there follows a brief description of how to use these facilities from the editor.



Assembler options are grouped into three separate requesters which can be called up by selecting one of three items on the assembler settings sub-menu; Control-, Options- .. and Optimisations... (Ctrl-1, 2 or 3 from the keyboard). Note that some assembler options cannot be selected from these requesters either because they are available as a separate Program menu command or because they are inappropriate to the integrated environment.

The Control requester provides control over the operation of the assembler, listing control and what type of file is produced. This requester also includes the path and name of your *main file*.

The main file is the source file upon which all assembly commands on the Program menu operate and is of use when creating a large program comprising of a number of source files included by the main file. Where no main file is specified (the default), assembly is carried out on the current project.

The Options requester gives access to a whole range of assembly options including processor type and enables various levels of source checking provided by the GenAm assembler.

Finally, Optimisations allows you to determine which instructions will be optimised by the assembler and whether you are notified of these during assembly via a warning message.

Note that the Restore command will lose any modifications you have made to all open assembler options requesters, effectively cancelling each requester and re-displaying the state they were in when originally activated.

Options files

Unlike all other Devpac settings, the contents of the three assembly options requesters is saved in a file separate from the editor settings. They are stored in an assembly options file called `GenAm.opts` which may be saved and loaded in much the same way as normal settings files.

When the editor starts up, it will search for `GenAm.opts` in the same places as `Devpac.prefs`; the project directory, the editor directory and `ENV:Devpac`. This gives you the ability to save separate options files for different projects by placing them in the appropriate drawer.

Options files have the additional benefit of being usable from a command line environment. GenAm will search for the `GenAm.opts` file before processing its command line arguments allowing you to specify default options in this file. These may then be overridden by further options on the command line.

Selecting Save from the Assembler sub-menu will save all assembly options to the current options file or create a default file in the editor directory if none has been loaded. You may save a number of options files in different places via the Save As... submenu command although files named anything other than `GenAm.opts` will not be loaded automatically by either the Devpac editor or the assembler. Icons for options files are taken from `def_opts` or `def_project`.

Options files may then be reloaded using the Load... command from the Assembler sub-menu. If you wish to re-load the current options file, you may wish to use the Last Saved command instead. Reset to Defaults may be used to restore all assembler options to their original state.

Miscellaneous Commands

The About box

If you select About... from the Project menu (also available via the Help key), a requester will appear giving various details about HiSoft Devpac, including its version number. You will also be told the name of the current project and its size both in lines and bytes.

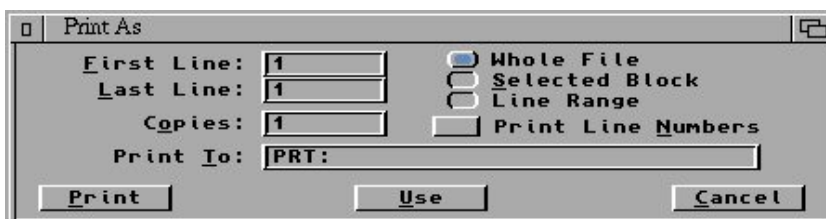
Pressing C or clicking on Continue will resume editing.

Printing

A range of printing options are available from HiSoft Devpac.

To simply print a project using the current settings you may press AP or select Print from the Project menu. A requester will appear if the printer cannot be found or an error occurs.

You may change the printer settings from the requester below which is accessed by selecting Print As or pressing Shift-AP.



the printer settings

Using the radio buttons, you may choose to print the entire project, the currently marked block or a selected range of lines entered via the first two number gadgets.

Optional line numbers may be added and the number of copies specified. Multiple copies will be separated by a form feed and tabs are expanded to the appropriate number of spaces.

The Print To gadget may be used to specify where the listing is sent. For most uses this should be set to PRT: which is the default printer although other AmigaDOS devices may be used. In order to select the type of printer or the serial or parallel port, do this via system preferences in the normal way (see your Amiga® documentation for further details).

It is possible to specify a file or pathname in order to re-direct printing to disk. This can be used as a method of converting tabs to spaces or adding line numbers to a file.

Read-Only projects

This feature allows a project to be temporarily locked, preventing any' accidental modifications. Blocks may still be marked or copied into the clipboard and the file can be saved but any command which would change the text will simply cause a Project is Read-Only error. Use Ctrl-W to turn this on or off for the current project.

Centre Window

Scrolls the current window so that the cursor is positioned as centrally as possible. Many commands such as Go to Bookmark or Find Next do this automatically although it is often a useful way of finding the cursor position when you get confused. The keyboard shortcut is *A* / or Ctrl-5 or Shift-5 on the numeric keypad.

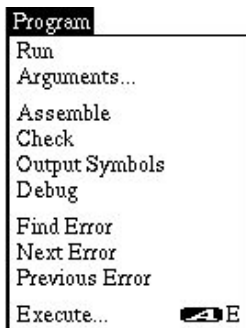
Select Font

By default, all windows use the currently selected system font. For Workbench 1.3 this is normally topaz 8 or 9 as specified in *preferences*. Workbench 2 may additionally set up any fixed width font from the *Font preferences* editor.

The Select Font... command on the Settings menu gives you the ability to choose a separate font for the Devpac editor. A requester will appear and selecting a font and size followed by OK will adjust all project windows to use your chosen font.

Assembling Programs

Having produced your assembly program using the editor you can then assemble it to memory or disk and run it. All program operations can be found on the Program menu with keyboard shortcuts using the Ctrl modifier key.



Although each command is covered in greater detail later in this chapter, here is an overview of the development process:

- Set up any options you wish to use via the Assembly sub-menu on the Settings menu.
- If you wish to produce a stand-alone program, select Assemble to Disk also on the Settings menu. Assemble to memory can only produce an executable format program.
- Select the Assemble command (using Ctrl-A). If the editor cannot find the GenAm assembler you will be given a chance to locate it. This can also be done via Resident Tools.

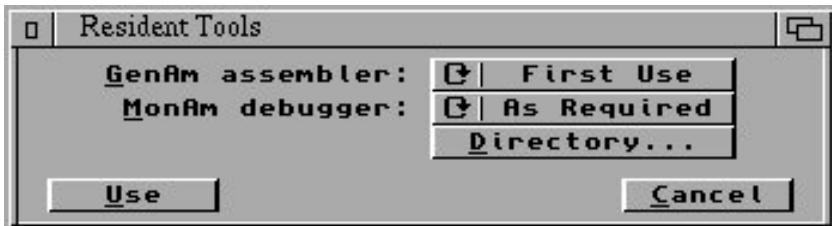
- Step through any errors using Next Error and Previous error (Ctrl-E and Shift-Ctrl-E).
- Once the program has successfully assembled, select Run from the Program menu (Ctrl-X). Arguments... may be selected if you wish to pass the program a command line.
- If the program did not function as intended, you may select the Debug (Ctrl-D) option in order to enter the MonAm debugger and trace through the program until you locate the problem.

A number of other assembly options are available which allow you to check that your program assembles correctly, generate a symbol file for faster assembly etc.

Remember that you may specify a particular file as your main source file, in that this will be assembled instead of the current project. This can be done from the Assembly Control requester. The main file may be loaded in the editor or on disk and will be read in accordingly.

Resident Tools

This command, on the Settings menu, lets you control how the Devpac programs are loaded into memory.



There are three different ways that the editor can load a resident tool. Firstly, by default, the assembler and debugger will be read into memory the first time that you select a command which requires them (First Use). Thus, if you do not use the debugger, for example, no extra memory will be taken up.

Alternatively you may specify that a tool should be loaded at all times. This will cause the program to be loaded when the editor starts up. If you are low on memory, the As Required option provides minimal memory usage, only loading a tool when absolutely necessary.

The Directory... gadget produces a file requester allowing you to locate the directory containing the GenAm and MonAm programs. This is also displayed if the editor comes to load a tool and cannot find it.

One final way of controlling resident tools is to load them externally by use of the AmigaDOS RESIDENT command. This way, the program will stay resident even when you quit HiSoft Devpac which will save on loading time if you have sufficient memory. To pre-load non-program files (such as include files or pre-assembled symbol files) we recommend copying them into the RAM disk.

Program menu

The commands on the Program menu are used to communicate with the other parts of the package, the assembler, the debugger and, not least, the program that you are developing.

Assemble

This command or its keyboard shortcut (Ctrl-A) assemble the main file specified in the Assembler Control requester which is accessed via the Settings menu. The rest of these settings are described at the start of the next chapter. If no main file has been entered the file in the current window will be assembled. If you have not even given your source code a name you can still assemble it, although the output file won't be useful.

If you haven't checked Assemble to Disk from the Settings menu then your file will be assembled to memory.

Whether the assembler, GenAm, is loaded from memory each time you invoke it is controlled using the Resident Tools item on the Settings menu.

Note that the assembler will automatically read include files from memory if they are being edited at the time - there is no need to save -any changes to include files that are being edited before assembly since they are read from memory, not disk.

Check

Check or Ctrl-C is just like assemble except that it does not produce output to memory or disk. If you know that your file contains errors this operation is slightly quicker than a normal assembly, even than an assemble to memory.

Output Symbols

This is used to produce a .gs file from an include file. The pre-assembled symbol table that is produced will then be loaded when you assemble a file that includes this file. Pre-assembly is described in detail in the next chapter.

Run

This command and its keyboard short cut (Ctrl-X) execute your program as if you had run it from the CLI. If you think there is any chance that your program will crash the machine, make sure that you save your source code before selecting Run.

If you have checked Assemble to Disk from the Settings menu, the Run command will also re-load your file from disk. Assembling to memory is a slightly faster operation.

If you have made changes to your text files since you last assembled you can still run your program in memory - but the executed program will not reflect the changes that you have made to the source.

Arguments

This command lets you set up the command line that is passed to your program when it is run. The command line is saved in the editor settings file, so that if you are working on a project that requires a command line, it will be set up as soon as you re-load the editor.

Debug

The **Debug** command (or **Ctrl-D**) command invokes the MonAm debugger which will automatically load your program, from disk or memory depending on your setting of the **Assemble to Disk** item on the **Settings** menu.

Execute

Execute (**A-E**) displays a requester for you to enter a CLI command (and its command line) that will then be executed - when the executed program terminates you will be returned to the editor.

One use for this command is to invoke the linker from within the editor. If you set up **Execute** to perform your link for you, you can subsequently invoke this using. **A-E** and **Return**.

Error commands

When an assembled program has errors in it the editor will report these errors and also remember a list of the error messages that occurred and the positions where they occurred.

If your assembly reported errors and you return to the editor, the cursor is placed on the line where the first error occurred with the error message displayed in the window title bar, regardless of whether this was the current window before the assembly started. If the first error is in a file that isn't currently loaded then this will be ignored and you will be positioned on the first error that is in a file that is loaded.

The error positions are treated internally in a very similar way to bookmarks - so you can insert, delete and move text and the error messages will still be displayed on the correct line.

Next Error

This command, with the shortcut **Ctrl-E**, will move to the next error in the list of errors regardless of current cursor position; it will switch windows if required. If there are no more errors the message no more errors will appear in the window title bar.

Previous Error

Previous Error (Shift-Ctrl-E) moves the cursor to the previous error in the error list, switching windows if required.

Thus Next Error and Previous Error can be used to navigate the entire list of errors - if you decide that you do not want to fix one mistake immediately, you can ignore it and then go back there using Previous Error.

Find Error

Find Error (Ctrl-F) moves to the next error starting from the cursor position in the current window. As such it works in a very similar way to the Jump to Error command of Devpac Amiga 2.

It has the disadvantage that, if you move the cursor off the current line, the next Find Error command may miss out some errors (if you moved to later in the file) or repeat errors (if you moved to earlier in the file).

Keyboard command summary

Project commands

⌘ N	New
⌘ O	Open
⌘ S	Save
⌘ A	Save As
⌘ P	Print
Shift- ⌘ P	Print As
Help	About
⌘ Q	Quit Devpac

Window commands

⌘ W	New Window
Shift - ⌘ W	Close Window
⌘ .	Activate Next
⌘ ,	Activate Previous
Shift - ⌘ >	Bring to Front
Shift - ⌘ <	Send to Back

Movement commands

← →	Character Left/Right
↑ ↓	Line Up/ Down
Shift- ←/→	Shift Left/Right
Shift-↑/↓	Shift Up/Down
Alt-←/→	Word Left/Right
Alt-↑ ↓	Shift Up/Down
Ctrl-←/→	Start/End of Line
Ctrl-↑/↓	Top/Bottom of File
Shift-Numeric ←/→	Scroll Left/Right
Shift-Numeric ↑/↓	Scroll Up/ Down
⌘ /, Shift-Numeric 5	Centre Window

Edit commands

Tab	Insert Tab
Return	Enter Line
Ctrl-Return	Insert Line
Ctrl- -	Join Lines
Ctrl-=	Split Line
Backspace/Del	
	Delete Character Left/Right
Shift-Backspace/Del	
	Delete to Start/End of Line
Alt-Backspace/Del	
	Delete Previous/Next Word
⌘ Backspace, Ctrl-Backspace, Ctrl-Y	
	Delete Line
Ctrl-U	Undelete Line
⌘ Z	Undo Line
Ctrl- W	Read-Only mode

Block commands

⌘ X, Shift-F5	Cut
⌘ C, Shift-F4	Copy
⌘ V, F5	Paste
⌘ Del, Shift- F3, Ctrl-Del	
	Erase
Esc	Unmark Block
F1	Mark Start
F2	Mark End
Shift-F1	Locate Start
Shift-F2	Locate End
F3	Save Block
F4	Duplicate Block
⌘ I	Insert file

Search commands

<i>A</i> F	Find
<i>A</i> R	Find & Replace
Ctrl-N	Find Next
Ctrl-P	Find Previous
Ctrl-R	Replace
<i>A</i> G	Go to Line
<i>A</i> 0-9	Go to Bookmark
Shift - <i>A</i> 0-9	Set Bookmark

Macro commands

Ctrl-[	Start Learning
Ctrl-]	Stop Learning
<i>A</i> M	Play Macro
Shift , <i>A</i> M	Repeat Macro

Program commands

Ctrl-X	Run
Shift-Ctrl- <i>A</i>	Arguments
Ctrl-A	Assemble
Ctrl-C	Check
Ctrl-O	Output Symbols
Ctrl-D	Debug
Shift-Ctrl-D	Debug without Program
Ctrl-F	Find Error
Ctrl-E	Next Error
Shift-Ctrl-E	Previous Error
Ctrl-1	Assembly Control
Ctrl-2	Assembly Options
Ctrl-3	Assembly Optimisations
<i>A</i> E	Execute
Ctrl-S	Set Settings

Chapter 3

The Assembler

Introduction

GenAm is a powerful, fast, full specification 68000/68040/68881 assembler, available instantly from within the editor or as a stand-alone program. It converts the text typed or loaded into the editor, optionally together with files read from disk, into a binary file suitable for immediate execution or linking. It can also produce a memory image for immediate execution from the editor.

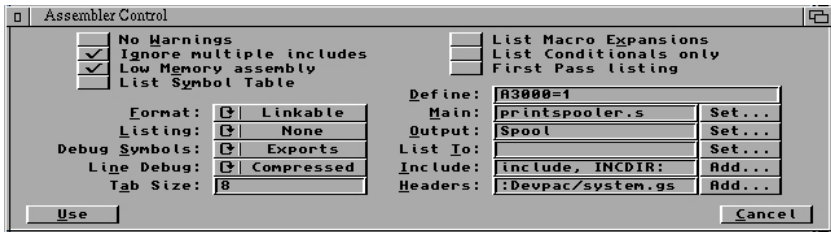
Invoking the Assembler

From the Editor

Before using the assembler you will probably need to set up the assembler's settings to reflect your preferences. This is achieved via the sub-menus from the Assembler item on the Settings menu.

The first three items on this menu display requesters that enable you to select the options. The first requester, Control, may also be selected using Ctrl-I and is described on the next page.

The Control requester



This requester is used to control the assembly process.

The No Warnings option suppresses the generation of assembler warnings.

Ignore multiple includes causes the assembler only to assemble an include file the first time that it is included. This leads to slightly faster assembly times and is most useful when using the operating system include files which may be called more than once. However you should not use this option if the multiple includes are each intended to generate code.

Low Memory Assembly lets you reduce the memory requirements of the assembler by causing it never to cache include files in memory. Under normal circumstances you should leave this button un-checked, and select it only if the assembler runs out of memory.

The List Symbol Table, List macro Expansions, List Conditionals only and First Pass listing options should be self explanatory. Generating a listing on pass one is only normally useful when debugging complex usage of conditional assembly.

The Format cycle gadget lets you select the output format produced by the assembler; either Executable, Linkable or S-Records. The differences between these are detailed later. Normally you will want to use Executable.

The Listing cycle gadget enables you to select an assembly listing. None will suppress the listing, Screen and Printer will direct it to the appropriate device. Disk will send the listing to a file based on the source filename but with the extension . LST. You may set your own file or device for the listing file using the List to item.

The Debug Symbols cycle gadget lets you choose which symbols to include ,in the executable. You normally include symbols in a file so that you can see them when using the debugger. The choice is:

None	Outputs no symbols
All	Outputs all symbols
Exports	When using linkable code only exports are output

Choosing Line Debug includes information about the code addresses corresponding to the line numbers in your program for use with the MonAm debugger. Two formats are available: Standard which uses LINE debug hunks that are compatible with the CodeProbe debugger that is supplied with SAS/C. Compressed uses HCLN hunks which need approximately one quarter of the space of LINE hunks but are not understood by CodeProbe. Beware that even using the compressed format increases the size of your program substantially. MonAm understands both formats.

Tab Size sets the size of tabs in listing files.

The Define item enables you to initialise the values of labels. This takes the form: label=value,label-value,... Thus:

A3000=1,STANDALONE=0

would set the label A3000 to have the value 1 and STANDALONE to have the value 0. Note that omitting the xxx causes the label to be set to the value 1.

Main sets the main file for assembly. If you are working on a project with multiple include files, you will want to set this up to be the main file to assemble. You may either type this in explicitly or set it via the File Requester by selecting the Set... gadget to the right of this item. If you leave this blank the file in the current window will be assembled when you select the Assemble item from the Program menu or use Ctrl-A.

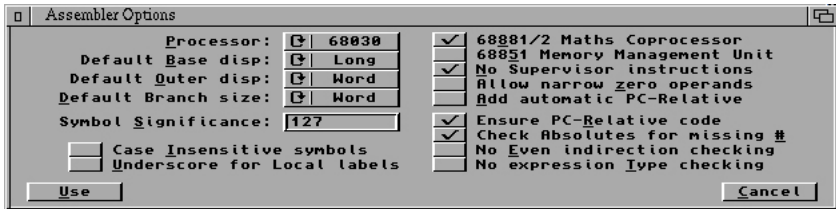
Output is used to override the default name for the assembler's output file which is the same as the main file but without an extension if executable, with a .O extension if linkable or an .MX extension if S-records are being produced. Even if you have specified an output filename with this option you will still need to ensure that the Assemble to Disk item is selected.

Include lets you set a list of directories that will be searched for include files. Typically you will set this up to point to the main directory for your operating system include files. You can add items to this using the File Requester by selecting the Add... gadget. To delete an entry you must use the keyboard.

Headers lets you set any pre-assembled header files that will be loaded before assembly begins. Such header files contain the symbol table information for macros and absolute labels and are produced using the Output Symbols (Ctrl-S) command from the Program menu. These files are described in detail later.

The Options requester

The second assembler settings requester is called Assembler Options and may be selected using Ctrl-2 as well as via the appropriate sub-menu. It looks like this:



The Processor cycle gadget lets you select for which main processor the assembler will generate code. If you are writing a program that is to run on all Amigas leave this as 68000. If you are writing a program specifically for the A3000 then you should select 68030.

The 68881 /2 Maths Coprocessor and 68851 Memory Management Unit check boxes enable the instructions for these coprocessors. Note that the 68851 should only be selected if you are generated code for a 68020/68851 combination; the 68030 and 68040 processor options automatically enable the MMU instructions for those particular chips. Equally it is not necessary to select the maths co-processor when using the 68040.

The Default Base disp and Default Outer disp cycle gadgets set the default sizes for these elements of the 68020 addressing modes. See below for a detailed discussion of these modes.

The Default Branch size option sets the default size for branch instructions. Using Word gives the same code as produced by 68000-only assemblers.

Use Case Insensitive symbols to select whether labels are case dependent or not. If Case Insensitive is checked, then Test and test would be treated as the same label, when not checked they are treated as different.

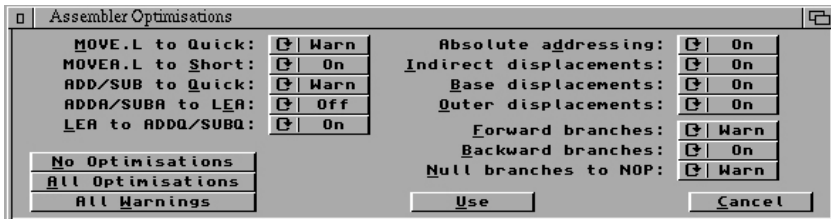
Symbol Significance lets you define the number of characters that will be considered when labels are compared. The default value is 127 characters, the maximum. The minimum value is 8.

The Ensure PC-relative code, Check Absolute values for missing # and No Even indirection checking options correspond to the CHKPC, CHKIMM and NOEVEN options when checked. These are described in detail below.

The Underscore for Local labels, No Supervisor instructions, Allow narrow zero operands, Add automatic PC-relative, No Even indirection checking and No expression Type checking items correspond to the LOCALU, USER, ALLOWZERO, AUTOPC, NOEVEN and NOTYPE options when selected. See below for details.

The Optimisations requester

The final assembler settings requester is that used for optimisations. The keyboard shortcut for this is Ctrl-3.



This requester controls the optimisations that will be made automatically by the assembler. Most of these can be set to one of On, Off or Warn, When Warn is selected the assembler makes the optimisation and issues a warning so that you can see where it is modifying your code. Switching an optimisation On causes it to be made without any warning.

The No Optimisations, All Optimisations and All Warnings gadgets can be used to switch off, on or warn of all optimisations at once. Note that the Forward branches gadget may only be set to Warn or Off.

The exact code transformations that these options perform are detailed below.

You will also need to check the setting of the Assemble to disk menu item on the Settings menu. When this is not selected assembly will be made directly to memory from where the program can immediately be Run or debugged via MonAm using the Debug item on the Program menu.

Having selected your required options you should select the Assemble menu item from the Program menu or press Ctrl-A to start the assembly. At the end of assembly press any key to return to the editor. If any errors occurred the cursor will be positioned on the first offending line.

If you wish to check the syntax of a program rather than actually assembling it to memory or disk, you can use the Check item from the Program menu - this is exactly like a normal assembly but it does not output any code. The keyboard shortcut for this is Ctrl-C.

The final form of assembly is to produce a pre-assembled header file so that you can use it via the Headers item on the Assembler Control requester. To produce a pre-assembled file select Output Symbols (Ctrl-S). Note that files that are used in this context may not contain code - only definitions of macros and constants. This is most useful for the operating system include files.

Assembly to Memory

To reduce development time GenAm can assemble programs to memory, allowing immediate execution or debugging from the editor. Such programs may self-modify if required as a re-executed program will be in its original state.

You should be careful when running from memory under the 68020 processor and above; these chips have an instruction cache and, if you modify an address that is already in the instruction cache, your new code will not be executed!

Stand-Alone Assembler

You can invoke the assembler from within the editor or from the Shell or CLI using the standalone assembler.

The standalone version of the assembler is called GenAm and, if it is called without a command line, you will be prompted for one conforming to the rules below; enter the options you want and press Return or press Return immediately to abort.

At the end of an assembly invoked from the editor, GenAm will pause, press any key to exit the program and return to the editor. If a command line has been supplied manually the assembler will not wait for a key at the end of the assembly as it assumes it has been run from a CLI or batch file.

Before GenAm processes its command line it looks for a file called GenAm.opts. This file can contain a number of lines of options in exactly the same format as the command line described below. This is exactly the format used by the editor for saving the assembler settings so that you can use the editor to set up your settings and just use the simplest form of command line for assembly.

Command Line Format

The GenAm command line consists of a series of options and the name of the main file that you wish to assemble. There may be options before and/or after the filename. You can use any of the long forms of OPT option names directly without any preceding character. These are described in detail later.

The main filename may be enclosed in quotes if you wish and can be preceded by the keyword FROM. This is compulsory in the unlikely event that your filename clashes with an OPT name. The main file's extension defaults to .S

The additional options that can be used from the command line are as follows; the corresponding OPT directive is shown in parentheses, where relevant. The latter options may be used on the command line without the OPT if you prefer.

- TO specify output filename.
- WITH specify a file that contains a list of options to be used. There may only be one WITH file per assembly and this option may only be used on the command line. Such files may be created using the editor's command for saving an assembler settings file. Be careful when using this as any main file that is set via the editor will be included in the WITH file.
- . (or QUIET) disable assembly messages.
- B no binary file will be created.
- C case insensitive labels (OPT NOCASE).
- D debug (OPT DEBUG).
- E allows labels to be set; assignments must be separated by commas. Labels will be set as if they were on line 2 of the main source file. See below for further details.
- H (or HEADER) specify the pre-assembled header files that are to be loaded before assembly starts. Multiple files may be separated with a comma.

- I (or INCDIR) specify include directories to be searched (follow immediately with path). These directories will be searched when the assembler is opening include files. These should normally be terminated with a slash.
- L Amiga® linkable code (OPT ALINK).
- L6 output Motorola S-records (OPT SREC).
- M use low memory (slower) assembly mode. See the section on integrated options in the previous chapter. Not the same as OPT M+.
- O specify output filename (which should follow *immediately* after O).
- P specify listing filename (which should follow *immediately* after P), defaults to source filename with extension of .LST. This may be any device.
- Q pause for key press after assembly.
- S include a symbol table at the end of the listing.
- T specifies the tab setting for listing. For example -T10 uses a tab setting of 10.
- V specify options as if they were specified using opt on the second line of the main source file.
- X use just exported labels in debugging (OPT XDEBUG).
- Z enable listing on pass 1. The information in the code field may be incorrect but this can be used to find mistakes when omitting an ENDC (OPT LIST 1). This is provided for backward compatibility; OPT TRACE IF can normally be used to find such errors more quickly.

The default is to create a executable binary file with a name based on the source file and output file type, no listing, with case sensitive labels.

Some examples of command lines:

```
genam test -b
```

assembles test.s with no binary output file.

```
genam test to ram:test -p
```

assembles test.s into a binary file ram:test and sends a listing file to test.lst.

```
genam test -l1dppar:
```

assembles `test.s` into linkable code with debug and a listing to the parallel port. (A listing to the serial port can be obtained by specifying `ser:` as the listing name).

`genam test alink debug -ppar:`

achieves the same effect.

`genam test with test-opts`

assembles the file `test.s` using the options contained in the file `test.opts`.

Defining Labels on the Command-Line

The `-E` option allows symbols to be defined at assembly time without having to change the source file. This option can be followed by one or more assignments of the form

`<symbol> = <expression>`

where `<symbol>` and `<expression>` follow the normal rules of the assembler and may contain values that have been defined previously. Multiple assignments must be separated by commas. If you omit the `= <expression>`, then the symbol will be assigned the absolute value 1. Such assignments occur as if they were the second line in the main source file, after any options.

Assembly Process

GenAm is a two-pass assembler; during the first pass it scans all the text in memory and from disk if required, building up a symbol table. If syntax errors are found on the first pass these will be reported and assembly will stop at the end of the first pass, otherwise, during the second pass the instructions are converted into bytes, a listing may be produced if required and a binary file can be created on the disk. During the second pass any further errors and warnings will be shown, together with a full listing and symbol table if required.

Assembly may be aborted by pressing `Ctrl-C`, although doing so will make any binary file being created invalid as it will be incomplete and should not be executed.

Return Codes

If using the CLI version of the assembler from batch- or make-files, you may exploit the codes the program returns. These are:

100+	initialisation failure
20	fatal error
10	error(s)
5	warning(s)
0	OK

Binary file types

There are three main types of binary files which may be produced by GenAm, for different types of applications. They are Arniga® executable, Arniga® linkable, and Motorola S-records.

GenAm can also assemble executable code directly to memory when using the integrated version allowing very fast edit-assembledebug-run times.

When producing linkable code, GenAm does not produce an executable file, but a file that needs to be processed by a linker to produce an executable file. An advantage of using this format is that your program can be linked with the output of a high level language compiler such as HiSoft BASIC. You can also use linkable code to split your assembly program into a number of modules.

Motorola S-records are the industry standard method of programming EPROMs and for standalone systems that use the 680x0 family of processors. The code produced is an ASCII file that runs at a particular address. As a result S-records are not suitable for executing directly on the Amiga® but are ready for downloading to an EPROM programmer or stand-alone system.

The type of executable file may be specified using the assembly options box, by using -l on the command line or OPT and then the format name within the source code. Such a directive must be placed before any code is generated and to avoid any confusion should be before any module or section directives.

The different possibilities are summarised in the table below:

Lx	OPT Name	Result	Extension
3	ALINK	ALINK linkable	.O
4	AMIGA	Amiga® executable	
6	SREC	Motorola S-records	.MX

Thus you can select linkable code from the assembler control requester or by using a command line like

```
test -13
```

or

```
test alink
```

or by including

```
opt ALINK
```

at the start of the file.

For the curious, option L values of 1, 2, 5 and 7 are used by other products in the Devpac family - ensuring source code compatibility across the range.

Output Filename

GenAm has certain rules regarding the calculation of the output filename, using a combination of that specified at assembly time (either in the Output: field in the assembler control options box or using the -o option on the command line) and the OUTPUT directive:

If an output filename is explicitly given at assembly time then

name=explicit filename

else

if the OUTPUT directive has not been used then

name=source filename + .O or .MX

else if the OUTPUT directive specifies an extension then

name=source filename + extension in OUTPUT

else

name=name in OUTPUT

Types of code

Unlike most 8-bit operating systems, but like most 16-bit systems, an executable program under AmigaDOS will not be loaded at a particular address but, instead, be loaded at an address depending on the exact free memory configuration at that time.

To get around the problem of absolute addressing the Amiga® file format includes *relocation information* allowing AmigaDOS to relocate the program after it has loaded it but before running it. For example, the following program segment

```
move.l      #string,a0
```

```
.
```

```
.
```

```
string      dc.b'Press any key',0
```

places the absolute address of string into a register, even though at assembly time the real address of string cannot possibly be known. Generally the programmer may treat addresses as, absolute even though the real addresses will not be known to him, while the assembler (or linker) will look after the necessary relocation information.

NOTE: **For certain programs, normally games or for S-record production an absolute start address may be required, for this reason the ORG directive is supported.**

The syntax of the language accepted by the assembler will now be described.

Assembler Statement Format

Each line that is to be processed by the assembler should have the following format:

<i>Label</i>	<i>Mnemonic</i>	<i>Operand(s)</i>	<i>Comment</i>
start	move.l	d0, (a0)+	store the result

Exceptions to this are comment lines, which are lines starting with an asterisk or semi-colon, and blank lines, which are ignored. Each field has to be separated from the others by white space - any number or mixture of space and tab characters.

Label field

The label should normally start at column 1, but if a label is required to start at another position then it should be followed immediately by a colon (:). Labels are allowed on all instructions, but are prohibited on some assembler directives, and absolutely required on others. A label may start with the characters A-Z, a-z, @ or underline (_), and may continue with a similar set together with the addition of the digits 0-9 and the period (.). Note that the @ character is allowed as the first character *except* when followed by a digit from 0-7 when it is taken as the start of an octal number.

Labels starting with a period are *local labels*, described later. Sequences of digits terminated by a \$ are also local labels. Macro names and register equate symbols may not have periods in them, though macro names may start with a period. By default the first 127 characters of labels are significant, though this can be reduced if required. Labels should not be the same as register names, or the reserved words SR, CCR, USP or any of the other special registers described under *Special Addressing Modes* below.

By default labels are case-sensitive though this may be changed. Some example legal labels are:

```
test, TEST, _test, -test.end, test5, _5test, @test
```

Some example illegal labels are:

```
5test, _&e, test>,
```

There are three reserved symbols in GenAm, all starting with two underline characters. These are `__LK`, `__RS` and `__G2`.

Mnemonic Field

The mnemonic (or opcode) field comes after the label field and can consist of 680x0 assembler instructions, assembler directives or macro calls. Some instructions and directives allow a size specifier, separated from the mnemonic by a period. Allowed sizes are `.B` for byte, `.W` for word, `.L` for long and `.S` for short. In addition floating point instructions may also have sizes of `.X` for extended, `.D` for double, `.P` for packed decimal and `.S` for short floating point. Some of the MMU instructions have a size of `.D` indicating a double long word (64-bits).

The specifiers which are allowed depend on the particular instruction or directive. GenAm is case-insensitive to mnemonic and directive names, so `Move` is the same as `move` and the same as `mOvE`, for example.

Operand Field

For those instructions or directives which require operands, this field contains one or more parameters, separated by commas. GenAm is case-insensitive regarding register names so they may be in either, or mixed, case.

Comment Field

Any white space not within quotation marks found after the expected operand(s) is treated as a delimiter for a start of the comment and will be ignored by the assembler.

Examples of valid lines

```
        move.l d0,(a0)+      comment is here
loop    TST.W  d0
lonely.label
    rts
* this is a complete line of comment
; and so is this
    indented: link a6,#-10 make room
a_string: dc.b 'spaces allowed in quotes' a string
```

Expressions

GenAm allows complex expressions and supports full operator precedence, parenthesis and logical operators.

Expressions are of two main types - *absolute* and *relative* - and the distinction is important. Absolute expressions are constant values which are known at assembly-time.

Relative expressions are program addresses which are not known at assembly-time as the AmigaDOS loader can put the program where it likes in memory. Some instructions and directives place restrictions on which types are allowed and some operators cannot be used with certain type-combinations.

Symbols used in expressions will be either relative or absolute, depending on how they were defined. Labels within the source will be relative, while those defined using the EQU directive will be the same type as the expression to which they are equated.

The use of an asterisk (*) denotes the value of the program counter at the start of the instruction or directive and is always a relative quantity.

Operators

The operators available, in decreasing order of precedence, are:

- monadic minus (-) and plus (+)
- bitwise not (~)
- shift left (<<) and shift right (>>)
- bitwise And (&), Or (!) and Xor (^)
- multiply(*) and divide (/)
- addition (+) and subtraction (-)
- equality (=), less than (<), greater than (>), inequality (<>) less than or equals (<=), greater than or equals (>=)

The comparison operators are signed and return 0 if false or -1 (\$FFFFFFFF) if true. The shift operators take the left hand operand and shift it the number of bits specified in the right hand operand; vacated bits are filled with zeroes.

This precedence can be over-ridden by the use of parentheses (and). With operators of equal precedence, expressions are evaluated from left-to-right. Spaces in expressions (other than those within quotes as ASCII constants) are not allowed as they are taken as the separator to the comment.

All expression evaluation is done using 32-bit signed-integer arithmetic, with no checking of overflow.

Note that | (vertical bar) may be used as a synonym for ! (or) and that != may be used as a synonym for <> (inequality).

Numbers

Absolute numbers may be in various forms:

- decimal constants, e.g. 1029
- hexadecimal constants, e.g. \$12f
- octal constants, e.g. @730
- binary constants, e.g. %01100010
- character constants, e.g. 'X'

\$ is used to denote hexadecimal numbers, % for binary numbers, @ for octal numbers and single ' or double quotes " for character constants.

The default base for numbers may be changed using the `RADIX` directive.

Character Constants

Whichever quote is used to mark the start of a string must also be used to denote its end and quotes themselves may be used in strings delimited with the same quote character by having it occur twice. Character constants can be up to 4 characters in length and evaluate to right-justified longs with null-padding if required. For example, here are some character constants and their ASCII and hex values:

"Q"	Q	\$00000051
'hi'	hi	\$00006869
"Test"	test	\$54657374
"it's"	it's	\$6974277C
'it"s,	it's	\$6974277C

Strings used in `DC.B` statements follow slightly different justification rules, detailed with the directive later.

Floating point constants

Floating point constants are only allowed as arguments for floating point instructions and `FEQU` directives. Such constants may either be expressed in hexadecimal or conventional decimal notation. Hexadecimal floating point constants should be preceded by `:` or `$`. The colon is the Motorola standard. When using hexadecimal the way in which the value is interpreted depends on the size used in the instruction. Thus

```
fmove.x #$400000008000000000000000, fpo
fmove.s #$40000000, fpo
fmove.d #:4000000000000000, fpo
fmove.p #$000000020000000000000000, fpo
fmove.s #2.0, fpo
```

all load the floating point, register `FPO` with the value 2.0.

For a description of the floating point formats see *Appendix E*.

Decimal numbers consist of one or more decimal digits followed by an optional fractional part consisting of a full stop (period) and an arbitrary number of decimal digits, optionally followed by an exponent consisting of the letter E or e and a signed decimal exponent. The maximum values allowed depend on the size of the appropriate instruction.

```
fmove.s #2, fp0
fmove. x #2E1, fp0
fmove.x #12345.678e -789, fp1
fmove.x #0.001, fp2
fmove.x # -1.2345E1234, fp3
```

The only operator that is allowed in floating point expression is the unary minus (-) operator.

Allowed Type Combinations

The table below summarises for each operator the results of the various type combinations of parameter and which combinations are not allowed. An R denotes a Relative result, an A denotes absolute and a * denotes that the combination is not allowed and will produce an error message if attempted.

	A op A	A op R	R op A	R op R
Shift operators	A	*	*	*
Bitwise operators	A	*	*	*
Multiply	A	*	*	*
Divide	A	*	*	*
Add	A	R	R	*
Subtract	A	*	R	A
Comparisons	A	*	*	A

Addressing Modes

The available 68000 addressing modes are shown in the table below. Please note that GenAm is case-insensitive when scanning addressing modes, so d0 and A3 are both valid registers.

Form	Meaning	Example
dn	data register direct	d3
An	address register direct	a5
(An)	address register indirect	(a1)
(An)+	address register indirect with post-increment	(a5)+
-(An)	address register indirect with pre-decrement	-(a0)
d An)	address register indirect with displacement	20(a7)
d(An, Rn.s)	address register indirect with index	4(a6, d4.L)
d.W	absolute short address	\$0410.W
d.L	absolute long address	\$12000.L
d(PC)	program counter relative with offset	NEXT(PC)
d(PC, Rn.s)	program counter relative with index	NEXT(PC, a2.W)
#d	immediate data	#26
n	denotes register number from 0 to 7	
d	denotes a number	
R	denotes index register, either a or d	
s	denotes size, either W or L, when omitted defaults to W	

When using address register indirect with index the displacement may be omitted, for example

```
move.l (a3, d2.1), d0
```

will assemble to the same as

```
move.l 0(a3, d2.1-), d0
```

The modes discussed above can be used regardless of the processor type. The following are additional modes that are only available when using a 68020 or later processor.

Extended Index Registers for 68020

Certain existing modes have been extended to support a scale on the index register as follows:

`exp(An,Xn<.size><*scale>)`

`exp(PC,Xn<.size><*scale>)`

If the above syntax is used then the expression must fit into 8 bits; if it is larger then the new modes `(bd,an,Xn)` / `(bd,PC,Xn)` should be used. Suppressed (Z) registers cannot be used with this syntax. See below.

New 68020 Modes

The new modes in their most basic form are:

<code>(bd,An,Xn)</code>	address register indirect with index (base displacement)
<code>([bd,An],Xn,od)</code>	memory indirect post-indexed
<code>([bd,An,Xn],od)</code>	memory indirect pre-indexed
<code>(bd,PC,Xn)</code>	program counter indirect with index (base displacement)
<code>([bd,PC],Xn,od)</code>	program counter indirect post-indexed
<code>([bd,PC,Xn],od)</code>	program counter indirect pre-indexed

Every item in the above is optional and within each set of brackets the item list may be in any order. In general the meaning of the syntax is that the processor takes the sum of any items in brackets and then performs an indirection (memory access) for each set of brackets.

For example, consider,

`move.w ([ $12.w, a1, d1 ], $24.w), d0`

and let us assume that `a1` has the value `$1230002` and `d1` has the value `$1234`. Then this will cause the processor to calculate `a1+d1+$1234` (giving `$1231248`) and fetch the long word value from that address. Assuming `$1231248` contains `$12345678` then `$24` will be added to this (giving `$1234569C`) and finally the word contents of `$1234569C` will be loaded into the least significant word of `d0`.

Depending on which items have been omitted, the assembler may change the choice of addressing mode to be more optimal. If you wish to have a particular mode with missing items then the item may be suppressed using Z-notation, i.e. specifying ZAn, or ZPC, as required. The elements described above are further detailed below:

bd - Base Displacement

This is an expression which may be relative or absolute, word or longword in size. The default size is *long*, but word may be forced by adding .W after the expression. The default size itself may be changed with the BDW and BDL options. If the base displacement is known on pass 1 the size can be optimised automatically by GenAm using opt 08+.

Xn - Index Register, with optional size and scale

This item has the general form Xn <.size> < *scale> where the size may be .W (the default) or .L. The scale must evaluate to 1, 2, 4 or 8.

od - Outer Displacement

This is an expression which may be word or longword in size but must be absolute. The default size is word, but long may be forced by adding L after the expression. The default size itself may be changed with the ODW and ODL options. If the outer displacement is known on pass I the size can be optimised automatically by GenAm using opt 09+.

New 68020 Syntax for Old Modes

The new syntax for the old modes is:

(d16,An)	equivalent to exp (An)
(d8,An,Xn)	equivalent to exp (An, Xn), though Xn may be scaled
(d16,PC)	equivalent to exp(PC)
(d8,PC,Xn)	equivalent to exp (PC, Xn), though Xn may be scaled

If any items are explicitly suppressed then a suitable new 68020 addressing mode will be used.

Ordering Rules

Any set of items within brackets may be ordered arbitrarily, though care should be taken if two address registers are specified; the leftmost register will be used as the base register, the rightmost as the index register. For example if the mode

```
(a3, isize, a2)
```

is specified the assembler will assume a3 is the base register and a2 will be sign-extended from 16-bits, as .W is the default index size.

Data Register Indirect

The 68020 allows data register indirection, by suppressing suitable items, but take care; the default size for index registers is word, so the line

```
move.l    (d3), d0
```

will actually be coded as

```
move.l    (0, za0, d3.w), d0 the (bd, An, Xn) form
```

which will indirect via the sign-extended value of d3; the likely correct line is

```
move.l    (d3.1), d0
```

Special Addressing Modes

CCR	condition code register
SR	status register
USP	user stack pointer

In addition to the above, SP can be used in place of A7 in any addressing mode, e.g. 4(SP, d3.w)

The data and address registers can also be denoted by use of the reserved symbols R0 through R15. R0 to R7 are equivalent to d0 to d7; R8 to R15 are equivalent to A0 to A7. In general we recommend sticking to the standard register names, but this option can be useful when porting code from other assemblers or to simplify tools which generate assembly language.

The registers above are available on all 68000 family processors. The following registers are only available on the The following registers are only available on higher processors and in the 68851 MMU. The use of some of these registers varies from chip to chip.

SFC, DFC	alternate function code registers
VBR	vector base register
CACR	cache control register
CHAR	cache address register
MSP	master stack pointer
ISP	interrupt stack pointer
CRP	CPU root pointer
SRP	supervisor root pointer
TC	MMU translation control register
TTO, TT1	translation control registers
MMUSR	MMU status register
ITTO , ITT1	instruction transparent translation registers
DTTO, DTT1	data transparent translation registers
DRP	DMA root pointer register
PCSR	MMU cache status register
AC	access control register
CAL	current access level
VAL	validate access level register
SCC	stack change control register
PSR	MMU status register
BAD0 - 7	breakpoint acknowledge data register
BAC0 - 7	breakpdint acknowledge control register

In general, user programs running under AmigaDOS should not use these registers since they are reserved for use by the operating system. Some low level programs may find it necessary to manipulate the cache control register if using self-modifying code; but these should call the Exec Cache Control routine rather than modifying the register directly.

Floating point registers

FP0-FP7	general purpose floating point registers
FPCR	floating point control register
FPSR	floating point status register
FPIAR	floating point instruction address register

The addressing modes used in conjunction with the floating point instructions are the same as those for the 'ordinary' instructions, although you should not that the floating point instructions always use at least one floating point register.

Calling the system maths libraries will automatically use a coprocessor if present. Whilst this is not nearly as fast as using the FPU directly it has the advantage of working on all systems.

Local Labels

GenAm supports local labels, that is labels which are local to a particular area of the source code. These are denoted by starting with a period and are attached to the last non-local label, for example:

```
len1    move.l    4(sp), a0
.loop   tst.b     (a0)+
        bne.s     .loop
        rts
len2    move.l    4(sp), a0
.loop   tst.b     -(a0)
        bne.s     .loop
        rts
```

There are two labels called `.loop` in this code segment but the first is attached to `len1`, the second to `len2`.

As the local labels are attached in this way, you must have at least one real label before the first local one.

If you wish to use global labels starting with a dot you may use `OPT LOCALU` to allow this and the underline character introduce local labels.

The local labels `.W` and `.L` are not allowed to avoid confusion with the absolute addressing syntax.

You may also use strings of decimal digits terminated by a \$ sign as local labels. This facility has been provided for compatibility with other assemblers; we recommend the use of form shown above as this makes programs much more readable.

Symbols and Periods

Symbols which include the period character can cause problems with GenAm due to absolute short addressing.

The Motorola standard way of denoting absolute short addresses causes problems as periods are considered to be part of a label, best illustrated by an example:

```
move .1      ExecBase.w, d0
```

where ExecBase is the absolute value 4 This would generate an undefined label error, as the label would be scanned as

ExecBase.w.

To work around this, the expression, in this case a symbol, may be enclosed in brackets, e.g.

```
move .1      (ExecBase).w, d0
```

though the period may still be used after numeric expressions, e.g.

```
move .1      4.w, d0
```

NOTE: **GenAm version 1 also supported the use of \ instead of a period to denote short word addressing and this is still supported in this version, but this is not recommended, due to the potential for \W and \L to be mistaken for macro parameters.**

Instruction Set

Word Alignment

All instructions with the exception of DC.B and DS.B are always assembled on a word boundary. Should you require a DC.B explicitly on a word boundary, use the EVEN directive before it. Although all instructions that require it are word-aligned, labels with nothing following them are not word-aligned and can have odd values. This is best illustrated by an example:

```
        nop    this will always be word aligned
        dc.b 'odd'
start
        tst.l (a0)+
        bne.s start
```

The above code would not produce the required result as start would have an odd value. To help in finding such instructions the assembler will produce an error if it finds an odd destination in a BSR or BRA operand. Note that such checks are not made on any other instructions, so it is recommended that you precede such labels with an EVEN directive if you require them to be wordaligned. A common error is deliberately not to do this, as you know that the preceding string is an even number of bytes long. All will be well until the day you change the string...

Instruction Set Extensions

The complete 68000-68040 instruction set is supported (depending on the processor selected) together with the 68881/68882 and 68551 coprocessors. A number of standard shorthands are automatically accepted as detailed below. A complete description of the 68000 instruction set can be found in the supplied pocket guide. Full details of the instructions for the other chips including syntax and addressing modes can be found in the M68000 Family Programmer's Reference Manual which is available from HiSoft.

Condition Codes

The alternate condition codes HS and LO are supported in BCC, DBCC and SCC instructions, equivalent to CC and CS, respectively.

Branch instructions

To force a short branch use `Bcc.B` or `Bcc.S`, to force a word branch use `BCC.W` or to leave to the optimiser use `BCC`. To use a 32 bit branch when 68020 or above code generation is in effect then use

`Bcc.L`.

When 68000/68008/68010 code generation is selected `Bcc.L` is interpreted as a `Bcc.w` with a warning for compatibility with GenAm 1. To cause `BCC.L` to be converted to `Bcc.W` regardless of the processor selected then use `OPT OLD` as described elsewhere.

A `BRA.S` to the immediately following instruction is not allowed but may be converted, with a warning, to a `NOP` using `OPT 07+` (see the options sub-section for details). A `BSR.S` to the immediately following instruction is not allowed and will produce an error.

DBcc Instruction

`DBRA` is accepted as an equivalent to `DBF`.

ILLEGAL Instruction

This generates the op-code word `$4AFC`.

LINK Instruction

If the displacement is positive or not even a warning will be given.

MOVE from CCR Instruction

This is a 68010 and upwards instruction, converted with a warning to `MOVE` from `SR` when 68000 only code is selected.

MOVEQ Instruction

If the data is in the range 128-255 inclusive a warning will be given. It may be disabled by specifying a long size on the instruction.

Assembler Directives

Certain pseudo-mnemonics are recognised by GenAm. These *assembler directives*, as they are called, are not (normally) translated into opcodes, but instead direct the assembler to take certain actions at assembly time. These actions have the effect of changing the object code produced or the format of the listing. Directives are scanned exactly like executable instructions and some may be preceded by a label (for some it is obligatory) and may be followed by a comment. If you put a label on a directive for which it not relevant, the result is undefined but will usually result in the label being ignored.

Each directive will now be described in turn. Please note that the case of a directive name is not important, though they generally are shown in upper case. The use of angled brackets (< >) in descriptions denote optional items, ellipses (...) denote repeated items.

Assembly Control

`END`

This directive signals that no more text is to be examined on the current pass of the assembler. It is not required.

`INCLUDE filename`

This directive will cause source code to be taken from a file on disk and assembled exactly as though it were present in the text. The directive must be followed by a filename in normal AmigaDOS format.

A drive specifier, directory and extension may be included as required, e.g.

`include df1:constants/header.i`

Include directives may be nested as deeply as memory allows and if any error occurs when trying to open the file or read it, assembly will be aborted with a fatal error. When using the integrated editor if an include file is loaded then this will be read direct from memory; there is no need to save it to disk before assembly.

If you have checked the Ignore multiple includes item in the Assembler Control requester or used the OPT INCONCE option, then attempts to include a file a second time will be ignored.

If no drive is specified, that of the main source file will be used when trying to open the file.

For maximum flexibility, GenAm allows a two ways of specifying where include files may be found without the need to specify the full pathname as in the example above.

First, the Assembler Control requester Include item (and its command line equivalent the -I option) lets you set directories that will be searched to find the include files via the . Second you can use the INCDIR directive itself to add to this path list.

Thus typically you use the assembler requester to set up the directory for the system includes and use the INCDIR directive for any files that are specific to this particular program.

NOTE: **The more memory the better, GenAm will read the whole of the file in one go if it can and not bother to re-read the file during pass 2.**

Pre-assembled files

When searching for include files GenAm also looks for a file with the same name as the include file but with any extension replaced with gs. This is assumed to be a pre-assembled symbol table file corresponding to that file name.

Such a file is produced using the Output Symbols option from the Program menu or by using the OPT GENSYM option. The .gs file that this produces contains the symbol table definitions for the absolute labels and macros that are defined by the include file. It also lists the files that the include file has included itself.

When the assembler loads a .gs file the labels and macros contained within it are added to the symbol table for the new assembly. If a definition is already present then it is ignored. Any subsequent references to this include file and the files that it includes will be ignored.

Thus if you Output Symbols from the system include file Intuition/Intuition.I this will generate intuition/intuition.gs. Your programs that use this include file will then load the .gs file and will assemble much more quickly. You can even delete the original include file if you wish (make sure you have a copy of the source first, though!).

If you know that your program is going to include a particular .gs file you can load it before assembly starts using the Headers requester from the Assembler Control requester or via the command line -H option.

Note that you cannot pre-assemble files that generate code. However the entire operating system includes may be preassembled in this way.

INCDIR pathnamelist

The INCDIR directive lets you specify directories that will be searched for include files as well as those specified via the -I command line flag or Include list in the assembler control requester.

The format for the pathnamelist is a list of items separated by commas or semi-colons and the directories should be terminated by a backslash. Pathnames that contain spaces should be enclosed in quotes.

INCBIN filename

This takes a binary file and includes it, verbatim, into the output file. Suggested uses include screen data, sprite data and ASCII files. The INCBIN directive uses the same method for finding files as the INCLUDE directive above.

The included data is forced to an even boundary, however the section counter is not forced to an even boundary after the include if the file is an odd number of bytes in length.

OPT option <,option>

There are a very wide range of options controlling all aspects of the assembly process; some may be set with their own option letter on the command-line, all may be set with an OPT directive within the source file, and most may also be set on the command-line using the -v option.

Devpac Amiga 2 and below only supported options denoted with an alphabetic character followed by + or -; however owing to the large number of options, these have been supplemented with keywords. The old format options are still accepted.

The options are as follows:

Processor selection

P=680x0 allows selection of processor type; main processor may be 68000, 68008, 68010, 68020, 68030, 68040 or 68332. Optional co-processors may be specified, separated by a / and may be any combination of 68881, 68882 or 68851. Specifying a main processor will de-select any current co-processor.

NOTE: **The processor selected using the assembler control requester will be inserted as if on line 2 of the program; thus overriding any selection on line 1 of the source code. Equally you should not use a 680x0 instruction on the first line of your program since the new processor will not have been selected yet!**

68020 Default Displacement Sizes

BDW makes any un-sized base displacements used in 68020 addressing modes word-sized; this will cause errors if you have any relocatable references as they cannot fit into a word.

BDL makes any un-sized base displacements long-sized

ODW makes any un-sized outer displacements word-sized

ODL makes any un-sized outer displacements long-sized

All of the above can be overridden on an individual basis by specifying .W or .L after the expression. In general we recommend the use of such explicit specifiers as it makes code more portable.

See below for automatic optimisation of long displacements into short ones.

Branch Control

OLD	ordinarily a BRA.L is converted into a BRA.W with a warning, unless you selected a 68020 or higher processor in which case it will generate a BRA.L, a 32-bit PC-relative branch. The use of this option will force BRA.Ls to always be converted to BRA.Ws, regardless of the processor type selected. Note that this option is not available in the integrated environment.
BRW	unsized branch instructions will default to BRA.W, unless optimisation type 1 is selected when it may be promoted to BRA.S.
BRB	unsized branch instructions will default to BRA.S; errors will be generated if the branch is out of range.
BRS	as above; included for Motorola compatibility
BRL	unsized branch instructions will default to BRA.L; P=68020 or P=68030 mode must be selected for this option to be valid

The default option is BRW.

Symbol Case Sensitivity

By default all symbols are case-sensitive, though this can be overridden on the command-line. The default length of symbol significance is 127 characters, the maximum.

CASE	symbols are case sensitive
NOCASE	symbols are case-insensitive
Cx	treat x characters as significant (x=8 -127).
Cx+	symbols are case-sensitive to x characters
Cx-	symbols are case-insensitive to x characters

Although it is unlikely to be useful, it is possible to use these options at any time in a source file and an unlimited number of times.

Listing Control

By default an assembly listing will show macro calls in the same form as those in the source.

MEX	expand macro calls in the listing
NOMEX	don't expand macro calls

By default there will be no symbol table listing, unless -S is specified on the command-line, or the option below is used:

SYMTAB	select a symbol table listing
NOSYMTAB	disable a symbol table listing
LIST1	enable assembly listing on pass one
NOLIST1	disable listing on pass one (default)

Pass one listing is not useful generally, because the data for the instructions may be wrong. It can be useful when tracking down mistakes with conditional assembly and macros.

TRACEIF	enable tracing of conditional assembly on pass one.
NOTRACEIF	disable this tracing (default).

The TRACEIF option is designed for finding mistakes in complex use of conditional assembly and as such is only for experienced assembly language users. Conditional assembly is described in a later section in this chapter. TRACEIF gives a list of only the IFxx, ELSExx and ENDC directives together with a display of the conditional assembly counter.

Debugging information

The AmigaDOS binary file format supports the inclusion of a SYMBOL hunk, which may be read by debuggers such as MonAm and can be extremely useful when debugging programs, since it allows you to use the labels from your program within the debugger. It also supports the idea of debug hunks. Whilst the contents of these are not defined by Commodore, there is a defacto standard for these that is supported by SAS/C, HiSoft BASIC 2, HighSpeed Pascal and Devpac 3. These enable the debugger to find the program counter corresponding to a source line and vice versa.

DEBUG	generates symbol hunks for all non-local labels
D	as above, included for Motorola compatibility.
NODEBUG	disable debug (default).
XDEBUG	generates symbol hunks only for exported labels.
LINE	generate LINE debug hunks for this file. This is the format that is supported by SAS/C and by Kink. If an error occurs when linking BLink will report the appropriate line number. This considerably increases the size of executable files however. 8 bytes are required for each line that generates code.

HCLN	generate HCLN (HiSoft Compressed Line Numbers) debug hunks for this file. These provide the same information as LINE hunks but normally require only 2 bytes of extra information per line that generates code.
NOLINE	these options both suppress the output of line
NOHCLN	debug (default).

Output - File Format

AMIGA, ALINK, , SREC

These select the output file format and must be before the first line of the source file that generates code; for further details please see the beginning of this chapter.

GENSYM

This causes GenAm to output a symbol table (or pre-assembled) file with extension .gs rather than a conventional output file. As such this command can only be used for include files that do not generate any code. See the section above on *Pre-tokenised files*.

Optimisation

GenAm is capable of optimising certain statements to faster and smaller versions. By default all optimising is off but each type can be enabled and disabled as required. This option has several forms:

01+ will optimise backward branches to short if within range, can be disabled with 01- .

02+ will optimise address register indirect with displacement addressing modes to address register indirect, if the displacement evaluates to zero. It can be disabled with 02- . For example

```

        move.l    next(a0),d3
will be optimised to
        move.l    (a0),d3
if the value of next is zero.
```


03+	will optimise absolute addresses to short-word addressing if in the signed 32 bit range \$FFFF8000 to \$7FFF inclusive.
04+	will optimise instructions of the form <code>MOVE.L #x, dn</code> to <code>MOVEQ</code> if <code>x</code> is in the range -128 to 127 inclusive.
05+	<code>ADD #x</code> and <code>SUB #x</code> instructions will be optimised to quick forms if <code>x</code> is in the range 1 - 8 inclusive.
06+	not strictly an optimisation; a warning will be issued for each forward branch that could be made short; this must be used in conjunction with option type 1.
07+	convert <code>BRA.S</code> to next instruction to <code>NOP</code> ; note that this instruction is not possible on the 680x0, so an error will be issued if this attempted without this optimisation.
08+	will optimise 68020 base displacements to the short form addressing if in the signed 32 bit range \$FFFF8000 to \$7FFF inclusive.
09+	will optimise 68020 outer displacements to the short form addressing if in the signed 32 bit range \$FFFF8000 to \$7FFF inclusive.
010+	will optimise <code>ADD #x, An</code> and <code>SUB #x, An</code> instructions to <code>LEA x(An), An</code> or <code>LEA -x(An), An</code> if this is possible but not in the case when an <code>ADDQ/SUBQ</code> instruction is preferable. This option is normally used in conjunction with 05+.
011+	will optimise <code>LEA x(An), An</code> or <code>LEA -x(An), An</code> instructions to <code>ADDQ.W #x, An</code> and <code>SUBQ.W #x, An</code> if this is possible.
012+	will optimise <code>MOVE.L #x, An</code> to <code>MOVE.W #x, An</code> if possible and if another optimisation has not been performed. This also optimises the correspond <code>ADD</code> , <code>SUB</code> and <code>CMP</code> instructions.
0+	will turn all optimising on
0-	will turn all optimising off
01- , 02- etc.	will disable the relevant optimisation

OW- will disable the warning messages generated by each optimisation, OW+ will enable them. OWn+/- (where n is 1-9) may be used to enable/disable a particular warning message.

If any optimising has been done during an assembly the number of optimisations made and bytes saved will be shown at the end of assembly.

Source Checking

The assembler has various ways of detecting possible programming errors, using these options:

CHKPC will force errors if any attempt is made to generate non-position-independent code

NOCHKPC disables the above (default)

CHKIMM will give errors if an absolute value is used in such a way that the assembler thinks it should be an immediate value, for example

```
and.b $df,d1
```

will generate the error # probably missing Can be overridden on an individual basis by specifying .W or .L after the expression. Note that omitting the # in front of 4 does not give an error when using the Amiga® formats so that lazy programmers do not need to change their references to ExecBase!

NOCHKIMM disables the above (default)

EVEN causes the assembler to check that the value of an indirection is not odd on non-byte sized instructions to avoid address errors (default), for example

```
move.l data2,d0
```

```
data1 ds.b 1
```

```
data2 ds.b 1
```

Do not confuse this option with the EVEN directive itself.

NOEVEN disables the above, useful if you are writing code to run only on a 68030.

Miscellaneous

INCONCE	causes multiple includes of the same file to be ignored. i.e. included only once. Whilst this speeds up the assembly of files that are 'protected' against multiple includes like the operating system header files, this will cause some files that need to be processed more than once. For example, rather than using a macro you might include a file a number of times to obtain two copies of the same routine or data.
NOINCONCE	causes include files to be re-scanned each time they are included (default).
AUTOPC	forces automatic PC addressing where possible; this is done on a lexical basis, not a value basis, and can be overridden individually by specifying .L, for example <pre>move.l test, a3</pre> will be changed to <code>test(pc), a3</code> Use of this option can significantly reduce program size and running time without a lot of extra typing. Please note however that it does not guarantee that the code generated will be position independent. We discourage the use of this option since it can easily cause confusion, particularly when using complex expressions. If you need to override the automatic use of PC mode then use the form <code>(expression).L</code> in a similar manner to that for short word addressing as described above under labels and periods. So the example above could be forced to use absolute addressing by using the following: <pre>MOVE.L (int_in).L, d0</pre>
NOAUTOPC	disables the above (default).
NOTYPE	Disables the type-checking of the expression evaluator which is capable of detecting incorrect type mixing; if you get an error <i>absolute not allowed</i> or <i>relative not allowed</i> and you are <i>sure</i> you know that you want to do what you're trying to do, then this will disable the checks.
TYPE	restores the type checking referred to above.
NOWARN	disables all warning messages.

WARN	enables warning messages (default).
USER	any privileged instructions used after this option will generate an error; useful for system programmers wishing to separate user and supervisor code spaces.
SUPER	permits privileged instructions to be used without errors (default).
LOCALU	changes the lead-in character for local symbols to be an underscore (_) instead of period; useful if you need to specify periods in external names.
LOCALDOT	changes the lead-in character for local symbols to be a period (.) (default).

Option Summary

Name	Default	Action	Old form
ALINK		linkable output	
ALLOWZERO		Allow narrow zero operands	
AMIGA		executable output	
AUTOPC		use PC relative addressing	A+
BDW		default base disp. to word	
BDL	*	defaults base disp. to long	
BRB		default branches to short	
BRL		default branches to long	
BRS		default branches to short	
BRW	*	default branches to word	
CASE	*	case-sensitive symbols	C+
Cx+	*	case-sensitive symbols	Cx+
Cx -		case-insensitive symbols	Cx -
CHKIMM		check immediate operands	I+
CHKPC		disallow non-PC addressing	P+
D		debug	D+

Name	Default	Action	Old form
DEBUG		debug	D+
EVEN	*	ensure indirections are even	E+
GENSYM		Generate GEN symbol table file	
HCLN		Generate compressed line numbers	
INCONCE		Process multiple includes only once	
LATTICE	*	Obsolete option to enable extensions	Y+
LINE		Generate standard line numbers	
LIST1		generate pass 1 listing	Z+
LOCALDOT	*	use periods for local labels	U-
LOCALU		use underscores for local labels	U+
MEX		expand macro calls	
NOALLOWZERO	*	disable narrow zero operands	
NOAUTOPC		disable automatically PC	A-
NOCASE		case-insensitive symbols	C-
NOCHKIMM	*	disable immediate checks	I-
NOCHKPC		disable PC-only checks	P-
NODEBUG	*	disable debug	D-
NOEVEN		disable indirec checks	E-
NOINCONCE	*	Re-process mul includes	
NOHCLN	*	no output of line numbers	
NOLINE	*	no output of line numbers	
NOLISTI	*	no pass 1 listing	Z-
NOMEX	*	don't expand macro calls	M-
NOSYMTAB	*	no symbol table listing	S-

Name	Default	Action	Old form
NOTRACEIF	*	don't trace conditionals	
NOTYPE		no type-checking	T-
NOWARN		no warning messages	W-
O+		enable all optimisations	O+
O-	*	disable all optimisations	O-
ODW	*	default outer disp. to word	
ODL		defaults outer disp. to long	
OLD		obsolete - treat BRA . L as BRA . W	
OW-		disable all opt. warnings	OW-
OWX+		enable an optimisation warning	OWX+
OWX -		disable an optimisation warning	OWX -
Ox+		enable an optimisation	Ox+
Ox-		disable a specific optimisation	Ox-
P=680x0 / 68xxx	68000	specify processor	
SREC		S-record output	
SUPER	*	privileged op-codes allowed	
SYMTAB		enable a symbol table listing	S+
TRACEIF		trace conditionals	
TYPE	*	enable type checking	T+
USER		privileged op-codes disallowed	
WARN	*	enables warning messages	
XDEBUG		specify extended debug	X+

Assembler Directives

`<label> EVEN`

This directive forces the program counter to be even, i.e. word-aligned. As GenAm automatically word-aligns all instructions (except `DC.Bs` and `DC.Bs`) it should not be required very often, but can be useful for ensuring buffers and strings are word-aligned when required.

`CNOP      offset,alignment`

This directive aligns the program counter using the given offset and alignment. An alignment of 2 means word-aligned, an alignment of 4 means long-word-aligned and so on. The alignment is relative to the start of the current section. For example,

`cnop 1,4`

aligns the program counter a byte past the next long-word boundary.

```
<label> DC.B            expression<,expression> ...
         <label> DC.W    expression<,expression>
...
         <label> DC.L    expression<,expression>
...
         <label> DC.X fp_const<fp_const> ...
         <label> DC.D fp_const<fp_const> ...
         <label> DC.S fp_const<fp_const> ...
```

These directives define constants in memory. They may have one or more operands, separated by commas. The constants will be aligned on word boundaries for `DC.W` and `DC.L`. No more than 128 bytes can be generated with a single `DC` directive.

`DC.B` treats strings slightly differently to those in normal expressions. While the rules described previously about quotation marks still apply, no padding of the bytes will occur and the length of any string can be up to 128 bytes.

Be very careful about spaces in `DC` directives, as a space is the delimiter before a comment.

For example, the line

```
dc.b    1,2,3 ,4
```

will only generate 3 bytes - the , 4 will be taken as a comment.

The DC.X, DC.D and DC.S directives generate floating point constants and are only available if you have selected a maths coprocessor.

```
<label> DS.B expression
<label> DS.W expression
<label> DS.L expression
```

These directives will reserve memory locations and the contents will be initialised to zeros. If there is a label then it will be set to the start of the area defined, which will be on a word boundary for DS.W and DS.L directives. There is no restriction on the size, though the larger the area the longer it will take to save to disk (except in the case of true BSS sections).

For example, all of these lines will reserve 8 bytes of space, in different ways:

```
ds.b 8
ds.w 4
ds.l 2
```

```
<label> DCB.B number<,value>
<label> DCB.W number<,value>
<label> DCB.L number<,value>
```

This directive allows constant blocks of data to be generated of the size specified. number specifies how many times the value should be repeated. If value is omitted then the default value, zero is used.

FAIL

This directive will produce the error `user error`. It can be used for such things as warning the programmer if an incorrect number of parameters have been passed to a macro.

MACHINE machine_number

This directive sets the processor for which code is generated and should be one of:

MC68000	MC68008	MC68010
MC68020	MC68030	MC68040
MC68832	CPU32	

The last two items are equivalent. Note that you can also use the OPT p= option which also allows co-processors to be selected.

OUTPUT filename

This directive sets the normal output filename though can be overridden by specifying a filename at the start of assembly. If filename starts with a period then it is used as an extension and the output name is built up as described previously.

RADIX radix

This directive sets the default base for number literals. radix may be one of 2, 4, 8, 10 or 16 and must be specified in decimal; expressions are not allowed.

The default is decimal (base 10). Two reasons for using this command are to enter tables in a non-decimal base and to assemble code that has been generated by a disassembler or other tool that emits non-decimal numbers without the appropriate prefix.

When using hexadecimal base (16) numbers must still start with a *decimal* digit. For example,

radix	16
dc. b	0ff
dc. b	ff

Here 0ff would have the value 255 whereas ff would refer to the label ff.

`__G2` (reserved symbol)

This is a reserved symbol that can be used to detect whether a Devpac family assembler (other than Devpac 1) is used. To test that a Devpac family assembler is being used to use the IFD conditional. The value of this symbol depends on the version of the assembler and is always absolute.

To ensure that the assembler facilities over and above those of Devpac Amiga 2 are available (for example RADIX) you should check that the lower 8 bits of `__G2` are at least 43 (don't ask why!). This does not guarantee that the 68030 and co-processor instructions are supported.

To check for which processor you are assembling look at bits 8-15: these give the last two digits of processor number. For example \$1E if you are producing code for a 68030.

To ensure that you are running on an Amiga® machine (rather than, say, an Atari ST) check that bits 16 to 23 are 1.

`__LK` (reserved symbol)

This is a reserved symbol that can be used to detect which output mode is specified. The value of this symbol is always absolute and one of the following:

3	Amiga® linkable
4	Amiga® executable
6	Motorola S-records

Other values are reserved for other members of the Devpac family.

Repeat Loops

It is often useful to be able to repeat one or more instructions a particular number of times and the repeat loop construct allows this.

```
<label> REPT  expression
```

```
    ENDR
```

Lines to be repeated should be enclosed within REPT and ENDR directives and will be repeated the number of times specified in the expression. If the expression is zero or negative then no code will be generated. It is not possible to nest repeat loops. For example

```
    REPT    512/4          copy a sector quickly
    move.l  (a0)+,(a1)+
    ENDR
```

NOTE: **Program labels should not be defined within repeat loops to prevent label defined twice errors.**

Listing Control

```
LIST
```

This will turn the assembly listing on during pass 2, to whatever device was selected at the 'start of the assembly (or to the screen if None was initially chosen). All subsequent lines will be listed until an END directive is reached, the end of the text is reached, or a NOLIST directive is encountered.

Greater control over listing sections of program can be achieved using LIST+ or LIST- directives. A counter is maintained, the state of which dictates whether listing is on or off. A LIST+ directive adds 1 to the counter and a LIST- subtracts 1. If the counter is zero or positive then listing is on, if it is negative then listing is off. The default starting value is -1 (i.e. listing off) unless a listing is specified when the assembler was invoked, when it is set to 0. This system allows a considerable degree of control over listing particularly for include files. The normal LIST directive sets the counter to 0, NOLIST sets it to -1.

If you would like a listing on pass -1, you can use

```
OPT      LIST1
```

For further details of pass one listing see the *Options* section.

```
NOLIST
```

This will turn off any listing during pass 2.

When a listing is requested onto a printer or to disk, the output is formatted into pages, with a header at the top of every page. The header itself consists a line containing the program title, date, time and page number, then a .line showing the program title, then a line showing the sub-title, then a blank line. The date format will be printed in the form DD / MM / YY, unless the assembler is running on a US machine, in which case the order is automatically changed to MM / DD / YY. Between pages a form-feed character (ASCII FF, value 12) is issued.

```
PLEN     expression
```

This will set the page length of the assembly listing and defaults to 60. The expression must be between 12 and 255.

```
LLEN     expression
```

This will set the line width of the assembly listing and defaults to 132. The value of the expression must be between 38 and 255.

TTL string

This will set the title printed at the top of each page to the given string, which may be enclosed in single quotes. The first TTL directive will set the title of the first printed page. If no title is specified the current include file name will be used.

SUBTTL string

Sets the sub-title printed at the top of each page to the given string, which may be enclosed in single quotes. The first such directive will set the sub-title of the first printed page.

SPC expression

This will output the number of blank lines given in the expression in the assembly listing, if active.

PAGE

Causes a new page in the listing to be started.

LISTCHAR expression<,expression> ...

This will send the characters specified to the listing device (except the screen) and is intended for doing things such as setting condensed mode on printers. For example, on Epson printers and compatibles the line

```
listchar 15
```

will set the printer to 132-column mode.

FORMAT parameter<,parameter> ...

This allows exact control over the listed format of a line of source code. Each parameter controls a field in the listing and must consist of a digit from 0 to 2 inclusive followed by a + (to enable the field) or a - (to disable it):

- 0 line number, in decimal
- 1 section name/number and program counter

2 hex data in words, up to 10 words unless printer is less than 80 characters wide, when up to three words are listed.

Label Directives

label EQU expression

This directive will set the value and type of the given label to the result of the expression. It may not include forward references, or external labels. If there is any error in the expression, the assignment will not be made. The label is compulsory and must not be a local label.

label = expression

Alternative form of the EQU statement.

Label EQU register

This directive allows a data or address register to be referred to by a username, supplied as the label to this directive. This is known as a register equate. A *register equate* must be defined before it is used.

label SET expression

This is similar to EQU, but the assignment is only temporary and can be changed with a subsequent SET directive. Forward references cannot be used in the expression. It is especially useful for counters within macros, for example, using a line like:

zcount set zcount+1

(assuming zcount is set to 0 at the start of the source). At the start of pass 2 all SET labels are made undefined, so their values will always be the same on both passes.

label REG register-list

This allows a symbol to be used to denote a register list within MOVEM instructions, reducing the likelihood of having the list at the start of a routine different from the list at the end of the routine. A label defined with REG may be used in expressions, with a warning; they have a value which is the same as that used in the MOVEM post-increment opcode.

Defining offsets

There are three different ways to define lists of constant labels without using explicit numbers that would need to be changed if you decided to add or delete an item near the front of the lists. The first is using the RS directives, the second using an OFFSET section and the last is using the CARGS directive. The first two methods provide the same functionality although OFFSET directives usually require more lines of code than RS directives. CARGS requires less typing than the other two methods but can not be used for items of sizes other than 2 or 4 bytes.

```
<label> RS.B expression
        <label> RS.W expression
        <label> RS.L expression
```

These directives let you set up lists of constant labels, which is very useful for data structures and global variables and is best illustrated by a couple of examples.

Let's assume you have a data structure which consists of a long word, a byte and another long word, in that order. To make your code more readable and easier to update should the structure change, you could use lines such as

```
                rsreset
d_next          rs.l  1
d_flag          rs. b  1
d_where         rs.l  1
```

then you could access them with lines like

```
move.l          d_next(a0),a1
move.l          d_where(a0),a2
tst.b           d_flag(a0)
```

As another example let's assume. you are referencing all your variables off register A6 (as done in GenAm and MonAm) you could define them with lines such as

```
Onstate      rs.b 1
Start        rs.l 1
end          rs.l 1
```

You then could reference them with lines such as

```
move.b  onstate(a6),d1
move.l  start(a6),d0
cmp.l   end(a6),d0
```

Each such directive uses its own internal counter, which is reset to 0 at the beginning of each pass. Every time the-assembler comes across the directive it sets the label according to the current value (with word alignment if it is .W or .L) then increments it according to the size and magnitude of the directive. If the above definitions were the first RS directives, onstate would be 0, start would be 2 and end would be 6.

RSRESET

This directive will reset the internal counter as used by RS.

```
RSSET  expression
```

This allows the RS counter to be set to a particular value.

```
__RS    (reserved symbol)
```

This is a reserved symbol having the current value of the RS counter.

```
OFFSET expression
```

This switches code generation to a special section to generate absolute labels. The optional expression sets the program counter for the start of this section (otherwise the value left over from the last OFFSET section will be used. No bytes are written to the disk and the only code-generating directive allowed is DS. Labels defined within this section will be absolute.

To return to ordinary code generation, use a suitable `SECTION` directive. See under the different output formats below.

Thus if the current section is `TEXT` then

```
        OFFSET
lab1    ds.w   1
lab2    ds.l   2
        SECTION TEXT
```

works in a similar way to

```
lab1    rs.w   1
lab2    rs.l   2
```

and would assign the same values to `lab1` and `lab2`.

CARGS `<#offset,>lab1.size-e,<lab2.size>...`

This directive is designed for accessing subroutine parameters that have been passed on the stack and as such it is very useful when interfacing with high-level languages.

This defines `lab1` to have the value given by `offset` . The value of `lab2` would then depend on the size used for `lab1`. If this was `.L` then it will be 4 more than `offset`; if it is `.W` or `.B` then it will be 2 more than `offset` . Subsequent labels are defined in a similar way. The default value for `offset` is 4 and the default size for labels is 2 bytes.

Here is an implementation of the C function `strcat` which appends one null terminated string to the end of another. Its first parameter in C is the original string and the second is the string to be added. As is usual in C, the second parameter is pushed on the stack, and then the first parameter. The assembly language code is

```
strcat      cargs    original.l,added.l
            move.l    original(sp),a0
findend     tst.b     (a0)+
            bne.s     findend
            subq.w    #1,a0          ready to replace null
            move.l    added(sp),a1
copylp      move.b    (a1)+,(a0)+
            bne.s     copylp
            rts
```

Thus original will have a value of 4 and added will have a value of 8 corresponding to their offsets on the stack after a jsr or bsr instruction has been used.

If you are using a language in which parameters are passed in 'Pascal order' where the first parameter is pushed on the stack first, then you will need to reverse the order of the arguments in the CARGS directive.

Also note that although in many ways the CARGS directive is equivalent to use the RSSET directive to the value of the offset expression followed by the equivalent RS directives for the labels, it differs in one very important respect. Using .B is exactly equivalent to using .W. This is because the instruction

```
move.b d0, -(sp)
```

will decrease the stack pointer by 2 and place the low byte of the register on the even address of the new stack pointer. Thus to access such a parameter on the stack, previously pushed parameters will be two bytes further up the stack.

Thus CARGS cannot be used for defining data structures that contain byte aligned data.

Floating Point Directives

Note that DC.S, DC.D and DC.X are really floating point directives but they are documented above with their integer cousins.

```
label FEQU.x constant
```

This directive will set the value and type of the given label to be a floating point constant of the given value. The constant may be specified in hexadecimal or decimal as described previously under expressions. Alternatively you may use a previously defined floating point constant,

The label is compulsory and must not be a local label.

Note that the size (. x) is compulsory and should be one of

.S	single precision
.D	double precision
.X	extended precision
.P	packed decimal
.W	word
.L	long

For example:

ten	fequ.x 10.0
two	fequ.s :40000000
million	fequ.x 1E6
minusmillion	fequ.x -million

FOPT option<,option> ...

This directive allows you to set the floating point co-processor identifier and the rounding and precision of the assembler's internal floating point calculations. The valid options are:

ID=<id> This sets the co-processor identifier. By default, this is 1 as used on the Amiga® 3000 and as recommended by Motorola. However for systems with more than one FPU you will need to set this.

ROUND=<type>

This is used to set the rounding method used by internal floating point operations. <type> should be one of:

N	round to the nearest
Z	round towards zero
P	round towards + infinity
M	round towards - infinity

These correspond to the RND portion of the FPCR mode control byte. The default value is N.

PREC=<type>

This is used to set the precision used by internal floating point operations. <type> should be one of:

X	extended precision
S	single precision
D	double precision

These correspond to the PREC portion of the FPCR mode control byte. The default value is X.

For example:

fopt	ID=2	set the co-proc ID
fopt	ROUND=Z	round towards 0
fopt	PREC=S	single precision

Conditional Assembly

Conditional assembly allows the programmer to write a comprehensive source program that can cover many conditions. Assembly conditionals may be specified through the use of arguments, in the case of macros, and through the definition of symbols in EQU or SET directives. Variations in these can then cause assembly of only those parts necessary for the specified conditions.

There are a wide range of directives concerned with conditional assembly. At the start of the conditional block there must be one of the many IF directives and at the end of each block there must be an ENDC directive. Conditional blocks may be nested up to 65535 levels.

Labels should not be placed on IF or ENDC directives as the directives will be ignored by the assembler.

IFEQ	expression
IFNE	expression
IFGT	expression
IFGE	expression
IFLT	expression
IFLE	expression

These directives will evaluate the expression, compare it with zero and then turn conditional assembly on or off depending on the result. The conditions correspond exactly to the 68000 condition codes. For example, if the label `DEBUG` had the value 1, then with the following code,

```

        IFEQ    DEBUG
logon   dc.b    'Enter a command:',0
        ENDC
        IFNE    DEBUG
opt     d+      labels please
logon   dc.b    'Yeah, gimme man:',0
        ENDC

```

the first conditional would turn assembly off as 1 is not EQ to 0, while the second conditional would turn it on as 1 is NE to 0.

NOTE: IFNE corresponds to IF in assemblers with only one conditional directive.

The expressions used in these conditional statements *must* evaluate correctly.

IIF exp statement

This directive can be used for pieces of conditionally assembled code that only consist of one line. IIF stands for Immediate IF. If the value of `exp` is non-zero then the given statement is assembled, otherwise it is ignored. No `ENDC` should be used in conjunction with this directive: For example,

```
IIF    BASIC    c1r.b    basic_flag(a6)
```

will cause the line

```
c1r.b    basic_flag(a6)
```

to be assembled if the variable `BASIC` has a non-zero value.

The statement part cannot contain a label field, but you may include a label before the IIF. For example

```
mary    IIF    john    equ 42
```

will set the value of the label mary to be 42 if the value of the label john is non-zero. If the expression evaluates to 0 then mary will have the value of the current program count as if the line

```
mary
```

had been included in the code. As a result it is generally not a good idea to use IIF to assign to variables, although it is suitable for ordinary program labels that are the targets of branch instructions.

```
IFD      label
IFND     label
```

These directives allow conditional control depending on whether a label is defined or not. With IFD, assembly is switched on if the label is defined, whereas with IFND assembly is switched on if the label is not defined. These directives should be used with care otherwise different object code could be generated on pass 1 and pass 2 which will produce incorrect code and generate phasing errors. Both directives also work on reserved symbols.

```
IFC 'string 1','string2'
```

This directive will compare two strings, each of which must be surrounded by single quotes. If they are identical then assembly is switched on, else it is switched off. The comparison is case sensitive.

```
IFNC     I string 1','string2'
```

This directive is similar to the above, but only switches assembly on if the strings are not identical. This may at first appear somewhat useless, but when one or both of the parameters are macro parameters it can be very useful, as shown in the next section.

ELSEIF

This directive toggles conditional assembly from on to off, or vice versa. ELSE can be used instead of ELSEIF although ELSEIF is the traditional Devpac name for this directive.

ENDC

This directive will terminate the current level of conditional assembly. If there are more IF s than ENDC s an error will be reported at the end of the assembly.

Macro Operations

GenAm fully supports extended Motorola-style macros, which together with conditional assembly allows you greatly to simplify assembly-language programming and the readability of your code.

A macro is a way for a programmer to specify a whole sequence of instructions or directives that are used together very frequently. A macro is first defined, then its name can be used in a macro call like a directive with up to 36 parameters.

label MACRO

This starts a macro definition and causes GenAm to copy all following lines to a macro buffer until an ENDM directive is encountered. Macro definitions may not be nested.

If the word MACRO is followed by .W or .L then when expanding the macro the program counter will be rounded up to an even boundary.

ENDM

This terminates the storing of a macro definition, after a MACRO directive.

MEXIT

This stops prematurely the current macro expansion and is best illustrated by the INC example given later.

NARG (reserved symbol)

This is not a directive but a reserved symbol. Its value is the number of parameters passed to the current macro, or 0 if used when not within any macro. If GenAm is in case-sensitive mode then the name should be all upper-case. \# may be used as a synonym for NARG.

Macro - Parameters

Once a macro has been defined with the MACRO directive it can be invoked by using its name as a directive, followed by up to 36 parameters. In the macro itself the parameters may be referred to by using the backslash character (\) followed by an alpha-numeric (1-9, A-Z or a-z) which will be replaced with the relevant parameter when expanded or with nothing if no parameter was given. There is also the special macro parameter \0 which is the size appended to the macro call and defaults to W if none is given. If a macro parameter is to include spaces or commas then the parameter should be enclosed in between < and > symbols; in this case a > symbol may be included within the parameter by specifying >>.

A special form of macro expansion allows the conversion of a symbol to a decimal or hexadecimal sequence of digits, using the syntax \<symbol> or \<\$symbol>, the latter denoting hex expansion. The symbol must be defined and absolute at the time of the expansion.

The parameter \@ can be useful for generating unique labels with each macro call and is replaced when the macro is expanded by the sequence - nnn where nnn is a number which increases by one with every macro call. It may be expanded up to five digits for very large assemblies.

A true \ may be included in a macro definition by specifying \\.

The abbreviation \# is equivalent to NARG giving the number of parameters that have been passed to the macro.

A macro call may be spread over more than one line, particularly useful for macros with large numbers of parameters. This can be done by ending a macro call with a comma then starting the next line with an & followed by tabs or spaces then the continuation of the parameters.

In the assembly listing the default is to show just the macro call and not the code produced by it. However, macro expansion listings can be switched on and off using the `OPT M` directive described previously.

Macro names are stored in a separate symbol table to normal symbols so will not clash with similarly-named routines, and may start with a period.

Macro Examples

Example 1 - Calling a library

As the first example, a common way of calling an Amiga® library routine is:

- save register A6
- load A6 from a library pointer
- do a JSR with offset
- restore A6

A macro to follow these specifications could be

```
call_lib      MACRO
               move.l a6, -(sp)      get lib pointer
               move.l \2, a6
               jsr     _LV0\1(a6)    call it
               move.l (sp)+, a6      restore
               ENDM
```

The directives are in capitals only to make them stand out: they don't have to be. If you wanted to call this macro to use the DOS function `Output` the code would be:

```
call lib      Output, _DOSBase
```

When this macro call is expanded, `\1` is replaced with `Output` and `\2` is replaced with `_DOSBase`. `\0`, if it occurred in the macro, would be `W` as no size is given on the call. So the above call would be assembled as:

```
move.l a6, -(sp)
move.l _DOSBase, a6      get lib pointer
jsr     _LV0Output(a6)   call it
move.l (sp)+, a6        restore a6
```

Example 2 - an INC instruction

The 68000 does not have the INC instruction of other processors, but the same effect can be achieved using an ADDQ #1 instruction. A macro may be used to do this, like so:

```
inc          MACRO
              IFC    ' ', '\1'
              fail    missing parameter!
              MEXIT
              ENDC
              addq.\0 #1,\1
              ENDM
```

An example call would be

```
inc.1  a0
```

which would expand to

```
addq.1  #1, a0
```

The macro starts by comparing the first parameter with an empty string and causing an error message to be issued using FAIL if it is equal. The MEXIT directive is used to leave the macro without expanding the rest of it. Assuming there is a non-null parameter, the next line does the ADDQ instruction, using the \0 parameter to get the correct size.

Example 3 - A Factorial Macro

Although unlikely actually to be used as it stands, this macro defines a label to be the factorial of a number. It shows how recursion can work in macros. Before showing the macro, it is useful to examine how the same thing would be done in a high-level language such as Pascal.

```
function     factor(n:integer):integer;
begin
    if n>0 then
        factor:=n*factor(n-1)
    else
        factor:=1
    end;
```

The macro definition for this uses the SET directive to do the multiplication $n * (n-1) * (n-2)$ etc. in this way:

```
* parameter 1=label, parameter 2='n'
factor.      MACRO
              IFND   \1
\1           set    1          set if not yet defined
              ENDC   \2
              IFGT
              factor \1,\2-1    work out next level
down
\1           set    \1*(\2)     n=n*factor(n-1)
              ENDC
              ENDM
* a sample call
              factor test,3
```

The net result of the previous code is to set test to 3! (3 factorial). The reason the second SET has (\2) instead of just \2 is that the parameter will not normally be just a simple expression, but a list of numbers separated by minus signs.

So it could assemble to

```
test          set    test*5-1-1-1
```

(i.e. $test * 5 - 3$) instead of the correct

```
test          set    test*(5-1-1-1)
(i.e.  $test * 2$ ).
```

Example 4 - Conditional Return Instruction

The 68000 lacks the conditional return instructions found on other processors, but macros can be defined to implement them using the \@ parameter. For example, a return if EQ macro could look like:

```
rtseq          MACRO
                bne.s  \@
                rts
\@
                ENDM
```

The \@ parameter has been used to generate a unique label every time the macro is called, so will generate in this case labels such as _002 and _017.

Example 5 - Numeric Substitution

Suppose you have a constant containing the version number of your program and wish this to appear as ASCII in a message:

```
showname      MACRO
               dc.b  \1, '\<version>',0
               ENDM
               .
               .
version        equ      42
               showname  <'Real Ale Search Program v'>
will expand to the line
               dc.b  'Real Ale Search Program v', '42',0
```

Note the way the string parameter is enclosed in <>s as it contains spaces.

Example 6 - Processor selection

Suppose you are writing a program that you intend to provide both A500 and A3000 specific versions. Say you use the label PROC30 with value 1 to indicate that you are producing the 68030 version and with a value of 0 for the A500 version then you could define macros like these:

* An extb.l instruction if available

```
extb.l        MACRO
               IFNE  PROC30
               opt   p=68030
               extb.l \1
               ELSE  p=68000
               opt
               ext.w  \1
               ext.l  \1
               ENDC
               ENDM
```

* Move 4 characters to memory using post decrement

```
move1      MACRO
            IFNE   PROC30
            move.w #'1\2\3\4',\5
            ELSE
            move.b #'1',\5
            move.b #'2',\5
            move.b #'3',\5
            move.b #'4',\5
            ENDC
            ENDM
```

Then an appropriate call would be:

```
extbl d0
```

which would expand to

```
extb.l d0
```

or

```
ext.w d0
ext.l d0
```

and

```
move1 F,R,E,D,(a0)+
```

would expand to

```
move.l #'FRED',(a0)+
```

or

```
move.b #'F',(a0)+
move.b #'R',(a0)+
move.b #'E',(a0)+
move.b #'D',(a0)+
```

Example 7 - Complex Macro Call

Suppose your program needs a complicated table structure which can have a varying number of fields. A macro can be written to only use those parameters that are specified, for example:

table-entry:

```
MACRO
    dc.b  .end\@- *      length byte
    dc.b  \1             always
    IFNC  '\2', ''
    dc.w  \2, \3         2nd and 3rd together
    ENDC  \4, \5, \6, \7
    dc.l
    IFNC  '\8', ''
    dc.b  '\8'          text
    ENDC
    dc.b  \9
.end\@
    dc.b  0
ENDM

* sample call
    table-entry          $42,,,t1,t2,t3,t4,
&                        <Enter name:>,%0110
```

This is a non-trivial example of how macros can make a programmer's life much easier when dealing with complex data structures. In this case the table consists of a length byte, calculated in the macro using \@, two optional words, four longs, an optional string, a byte, then a zero byte. Note the use of the macro continuation character &.

The code produced in this example would be

```
    dc.b  .end_001
    dc.b  $42
    dc.l  t1,t2,t3,t4
    dc.b  'Enter name:'
    dc.b  %0110
.end_001    dc.b  0
```

Output File Formats

GenAm is very flexible in terms of output file formats. These are detailed in this section together with notes on the advantages and disadvantages of each. Certain directives take different actions, depending on what output file format is specified.

The exact details of using each format will now be described.

Executable Files

This type of file is directly executable, for example by doubleclicking on its icon from the Workbench or by typing its name in the CLI. The file may include multiple sections, relocation information and/or symbolic information. These files normally have no extension.

Advantages reduced development time for all but the largest programs.

Disadvantages messy if more than one programmer.

Linkable Files

When writing larger programs, or when writing assembly language modules for use from the high-level language, you need to generate a linkable file. The AmigaDOS linker format is supported by the majority of high-level languages for the Amiga® and normally have the extension of .O or, OBJ.

Advantages great degree of freedom.

Disadvantages library format means selective library linking can be slow.

S-records

Motorola S-records are the only format that produces absolute code directly.

Advantages Can be sent to most EPROM programmers directly. Easy to transmit to other systems because they are pure 7-bit ASCII characters.

Disadvantages Can't run on the Amiga® 'as is' and are approximately twice the size of the equivalent executable file. If you are writing your own communication routines, it may be best to use executable files as your starting point.

Choosing the Right File Format

If you are writing entirely in assembly language then the normal choice has to be executable - it is fast to assemble, no linking required, and allows assemble to memory for decreased development time.

If you are writing a larger program, say bigger than 32k object, or writing a program as a team, then *linkable* code often makes most sense.

Output File Directives

This section details those directives whose actions depend on the output file format chosen. The file format itself can be selected by one of the following methods: command line options using GenAm directly from the CLI; clicking on the radio buttons in the Assembly Options dialog box from the editor; or with the OPT L directive at the beginning of the source file.

Sections

SECTION string<,type>

This defines a switch to the named section and optionally defines its type. A program may consist of several sections which will be concatenated together with other sections of the same name in the final executable file. This will use the supplied string and type as the section name and type of this section (or hunk) respectively.

Each can be up to 32 characters long and should not include tabs or spaces or be enclosed in quotes. The casing of the name is significant. You may have many SECTION directives in your program.

The type can be one of the following (in upper or lower case):

CODE	code section, public memory
CODE_F	code section, fast memory
CODE_C	code section, chip memory
DATA	data section, public memory
DATA_F	data section, fast memory
DATA_C	data section, chip memory
BSS	BSS section, public memory
BSS_F	BSS section, fast memory
BSS_C	BSS section, chip memory

CODE sections are used for executable instructions, DATA sections for initialised data (constants), and BSS for un-initialised data. BSS sections have the advantage that they take no disk space - only the length of the BSS section is stored. If you define a section to be BSS you can only use the DS directive to produce code - any other code generating instructions will cause a binary file I/O error.

Data and BSS sections that are called `__MERGED` are treated specially by the linker; they are merged together into one section. This, coupled with the BLink reserved symbol `__LinkerDB`, enables both pre-initialised and un-initialised data references to be made via a single global address register. See the BLink section for more details. Do not use `__MERGED` as the name of a CODE section.

NOTE: Do not use types requesting Chip or Fast memory, with versions prior to version 2.3 of the ALINK linker - these versions will crash!

CODE/DATA/BSS

These directives are supported for compatibility with the Amiga® Macro Assembler. They are the same as specifying the directive as the section name and type.

`IDNT      string`

This will use the supplied name as the hunk unit name for this section. It can be up to 32 characters long and should not include tabs or spaces or be enclosed in quotes.

`MODULE string`

This is a synonym for the IDNT directive.

Imports & Exports

With both linkable types of program it is crucial to be able to import and export symbols, both relative symbols (i.e. program references) and absolute symbols (i.e. constants). The AmigaDOS linker format does not distinguish between these types; however by specifying the type when importing, the assembler can type check, often finding programming errors that would otherwise be missed.

```
XDEF      export<,export>...
```

This defines labels for export to other files. If any of the labels specified are not defined an error will occur. It is not possible to export local labels.

NOTE: This directive is ignored if you are generating directly executable files.

```
XREF      import<, import>...  
XREF. L import <,import >...
```

This defines labels to be imported from other files. If any of the labels specified are defined an error will occur. The normal XREF statement should be used to import a relative label (i.e. program reference), while XREF.L should be used to import absolute labels (i.e. constants). If you do not type your imports you should turn type-checking off using OPT T- .

Importing a label more than once will not produce an error.

NOTE: This directive is ignored if you are generating directly executable files.

```
COMMENT commentstring
```

This directive is ignored at present.

ORG expression

This will make the assembler generate position-dependent code and set the program counter to the given value. Normal AmigaDOS programs do not need an ORG statement even if position-dependent. More than one ORG statement is allowed in a source file but no padding of the file is done.

This directive is not allowed for linkable code.

Using Imports in Expressions

Executable code does not allow or require the use of imports but inter-section references are subject to the restrictions described below.

Only one import may be used in each expression; however, they may be added to an arbitrarily complex expression, so long as this lexically precedes it, for example:

```
move.l 3+(1<<count+5)+import
```

There are a number of different sorts of possible imports as shown below:

Name	Example
PC-word	move.w import(pc), a0 bsr import
PC-byte	move.b import(pc, d0) bsr.s import
byte	move.b #import, d0
word	move.w import(a3), d0
long	move.l import, d0
word base relative	move.l _import(a4)
byte base relative	move.l import(a4, d0), d0

Note that a reference to a symbol in a different section is regarded as an import and subject to the above rules, except that PC-relative inter-section references are not supported.

The base-relative facilities allow references to imports and other sections to be word offsets, to allow such things as

```
move.l _symbol(a4), d0
```

where `_symbol` is a relative import, which, strictly speaking, is nonsense. However this is converted to

```
move.l _symbol-_LinkerDB(a4), d0
```

`__LinkerDB` is a symbol created by the linker. See the BLink section for further details of the memory map.

Motorola S-records (SREC L6)

S-records are a standard way of transferring binary images between machines, using 7-bit ASCII codes only. It is particularly useful for uploading data to EPROM programmers.

The S-record file produced by the assembler is of the following format:

```
S0          module name
             <for each section>
S1/2/3      data
S9/8/7      execute address
```

The file may be split into low and high bytes (or 4 if generating code for machines with 32-bit buses) if required by the use of the `SRSplit` utility, described in Chapter 6.

S1/S2/S3 records are produced for the data according to whether the address is a 16, 24 or 32 bit value respectively. Up to 28 data bytes per line are generated. The execute address is taken as the first `ORG` in the program, with an S9/S8/S7 as appropriate to the value.

The individual S-records contain 5 fields, mostly in the form of ASCII hex bytes as follows:

- type (2 bytes) Sx where x is the type of the record (as above)
- count (2 bytes) The number of address, data and checksum bytes remaining on this line
- address (4,6 or 8 bytes) the address of this data

- data (varies) the actual data, upper-case hex (2 for each byte)
- checksum (2 bytes) checksum of everything (taken as bytes) except the type

The default extension is .MX.

SECTION name<,offset>

If off set is specified then the section will be assembled to run at the address specified in the following ORG (as normal) but the addresses contained within the S-records themselves will start at the off set address. This is useful for writeable data areas that will initially be in EPROM and are copied into RAM at startup, or for the situation where a PROM programmer requires the data to be uploaded to a particular address.

ORG address

Should always follow a SECTION directive. The first ORG in a non BSS section is taken as the execute address. Using more than one ORG per section is at your own risk; it is your responsibility to put the code in the correct place if you intend executing it.

Directive Summary

Assembly Control

END	terminate source code
INCLUDE	read source; file from disk
INCBIN	read binary file from disk
OPT	option control
EVEN	ensure PC even
CNOP	align PC arbitrarily
DC	define constant
DS	define space
DCB	define constant block
FAIL	force assembly error
RADIX	set number base

Repeat Loops

REPT	start repeat block
ENDR	end repeat block

Listing Control

LIST	enable listing
NOLIST	disable listing
PLEN	set page length
LLEN	set line length
TTL	set title
SUBTTL	set sub-title
PAGE	start new page
LISTCHAR	send control character
FORMAT	define listing format

Label Directives

EQU	define label value
EQUR	define register equate
SET	define label value temporarily
REG	define register list
RS	reserve space
RSRESET	reset RS counter
RSSET	set RS counter
CARGS	define parameter labels
OFFSET	define offset table

Floating Point Directives

FEQU	define floating point constant
FOPT	floating point options

Conditional Assembly

IFEQ	assemble if zero
IFNE	assemble if non-zero
IFGT	assemble if greater than
IFGE	assemble if greater than or equal to
IFLT	assemble if less than
IFLE	assemble if less than or equal to
IFD	assemble if label defined
IFND	assemble if label not defined

IFC	assemble if strings same
IFNC	assemble if strings different
ELSEIF	switch assembly state
ENDC	end conditional
IIF	immediate IF

Macros

MACRO	define macro
ENDM	end macro definition

Output File Directives

MODULE	set hunk unit name
IDNT	SAS/C synonym for the above
SECTION	switch section
CSECT	SAS/C switch section directive
XDEF	define label for export
XREF	define label for import
COMMENT	send linker comment
ORG	set absolute code generation
TEXT	abbreviated section commands
DATA	
BSS	
CODE	

Reserved Symbols

NARG	number of macro parameters
__G2	internal version number
__RS	RS, counter
__LK	output file type

Chapter 4

The Debugger

Introduction

Programs written in assembly language are particularly errorprone; even a slight coding mistake can result in the entire machine crashing since you are programming at such a low level.

These programming mistakes (known as *bugs*, after a spider that was found crawling around the core memory of one of the early computers) can range from the trivial, such as a missing CR in a printout, through the usual (an incorrect result) to the very serious where the computer crashes because you have used the wrong register or corrupted the system memory (like that spider).

To help you find and correct all forms of bugs, Devpac Amiga includes a debugger, MonAm. MonAm is a powerful symbolic debugger and disassembler which lets you examine programs and memory, execute programs an instruction at a time and trap processor exceptions caused by programmer error.

Although MonAm is a *low-level* debugger, displaying such things as 680x0 instructions and registers, it can also be used for debugging programs written with any compiler that generates machine-code output. If the compiler has the option to output the symbols into the executable file then you will see your procedure and function names within the code; you can even view your original source code and step through it, if the package that produced the code has line number debug support.

MonAm uses its own screen (in the Amiga® sense), so if you are debugging a program with windows your program will not be sent re-draw messages whilst you are using the debugger. Many other Amiga® debuggers do send these messages, which can be very confusing.

MonAm Concepts

Here is a swift look at the concepts behind MonAm; it is a good idea to read this section before moving on to the next sections, even if you are an experienced programmer..

Exceptions

MonAm employs the 680x0 processor *exceptions* to stop runaway programs and to single-step, so at this point it would be useful to explain them and detail what normally happens when they occur on an Amiga.

While using the 680x0 processors, there are various types of exception that can occur, some deliberately, others accidentally. An exception is a special condition that takes priority over normal processing - it might be an interrupt from an external device, an illegal instruction, an address error, a co-processor violation or a number of other pre-defined events.

When an exception occurs the processor's context is saved on the supervisor stack and execution is then transferred to any one of 256 different addresses, held in the *exception table* (on the 68010 upwards, the address of the start of this table is held in the *vector base register*, or VBR). This table is set up by the Amiga's operating system so that an exception effectively transfers control to Exec, which is part of the Amiga's operating system.

The operating system then looks to see if the task that was running when the exception occurred has installed an exception handler i.e. the task wants to handle exceptions itself. If it has, control is passed to that exception handler; this is how MonAm traps exceptions because MonAm has attached such an exception table to the task that it has executed.

Unfortunately, there are a few exceptions that MonAm cannot trap because Exec does not pass them on - in these cases the operating system does what it normally does in the absence of an exception handler, it produces a Software Error alert (the dreaded Guru).

MonAm actually uses two of the exception vectors itself, one to set breakpoints in programs and the other to allow single-stepping.

The various forms of exceptions, their usual results, and what happens when they occur with MonAm active is shown in the following table, which is a summary of the exception table. Note that the first 64 vectors are defined by Motorola:

Exception no.	Exception	MonAm active
0	reset initial interrupt stack pointer	not trapped
1	reset initial program counter	not trapped
2	bus error	trapped
3	address error	trapped
4	illegal instruction	breakpoint
5	zero divide	trapped
6	CHK instruction	trapped
7	TRAPV instruction	trapped
8	privilege violation	trapped
9	trace	single-step
10	line 1010 emulator	trapped
11	line 1111 emulator	trapped
12	reserved	trapped
13	co-processor protocol violation	trapped
14	format error	guru
16-23	reserved	trapped
24	spurious interrupt	guru
25-31	level x interrupt autovector where x=26-exception no.	not trapped
32-47	trap #0 to #15	trapped
48	FPCP branch or set on unordered condition	trapped
49	FPCP inexact result trapped	trapped
50	FPCP divide by zero	trapped
51	FPCP underflow	trapped
52	FPCP operand error	trapped

Exception no.	Exception	MonAm active
53	FPCP overflow	trapped
54	FPCP signalling NAN	trapped
55	reserved	trapped
56	MMU configuration error	guru
57	68851 illegal operation	guru
58	68851 access level violation	trapped
59-63	reserved	trapped
64-255	user defined vectors	not trapped

The causes of the above exceptions (and how best to recover from them) are given at the end of this section.

Front Panel Display

When MonAm is invoked it displays a *Front Panel* showing registers, memory, source code and instructions. The name Front Panel stems from the type of panels that were mounted on mainframe and mini computers to provide information on the state of the machine at a particular moment, usually through the use of flashing lights. These lights represent whether or not particular flip-flops (electronic switches) within the computer are open or closed; the flip-flops that are chosen to be shown on this panel are normally those that make up the internal registers and flags of the computer thus enabling programmers and engineers to observe what the computer is doing when running a program.

These were *hardware* front panel displays; what MonAm provides you with is a *software* front panel - the code within MonAm works out the state of the computer and then displays this information on the screen.

The MonAm display consists of a number of windows through which you can view the 680x0 registers, a disassembly of your program, your program's source code or a portion of memory - you choose what you want in each window (within certain limitations). The layout of MonAm's front panel is shown on the next page.

```

MonAm version 3.08 Copyright © 1997 HiSoft

1 Registers
d0 = 00000001 ***[] a0 = 002A4E29 0A 0000 0000 0000 0000 0000 00 00*****
d1 = ABCDABCD <I<I a1 = ABCDABCD ** **** ** ** ** ** ** ** ** ** ** 
d2 = ABCDABCD <I<I a2 = ABCDABCD ** **** ** ** ** 
d3 = ABCDABCD <I<I a3 = ABCDABCD ** **** ** ** ** 
d4 = ABCDABCD <I<I a4 = ABCDABCD ** **** ** ** ** 
d5 = ABCDABCD <I<I a5 = ABCDABCD ** **** ** ** ** 
d6 = ABCDABCD <I<I a6 = ABCDABCD ** **** ** ** ** 
d7 = ABCDABCD <I<I a7 = 002B340C 002A 4F28 0000 1050 002B 2158 002B *0(*P+IX*+
sr = 0008 UI N
pc = 002D20F8 movem.l d0/a0,-(a7) ;002B3408 ABCD ABCD 002A 4F28

2 Disassembly pc
002D20F8 >movem.l d0/a0,-(a7)
002D20FC clr.l returnMsg
002D2102 suba.l a1,a1
002D2104 movea.l 4,w,a6
002D2108 jsr -$126(a6)
002D210C movea.l d0,a4
002D210E tst.l $A0(a4)
002D2112 beq.s fromWorkbench
002D2114 movem.l (a7)+,d0/a0
002D2118 bra.s end_startup
002D211A fromWorkbench lea $5C(a4),a0

3 Memory
002D20F8 48E7 8080 Hg[]
002D20FC 42B9 002D B+--+
002D2100 2162 9369 1b[]
002D2104 2078 0004 ,x-[]
002D2108 4EAE FEDA N9p[]
002D210C 2840 4AAC (8J~
002D2110 00AC 6706 *-g[]
002D2114 4CDF 0101 L8[]
002D2118 6024 41EC $A1
002D211C 005C 2C78 ^,x,
002D2120 0004 4EAE *[]N9
002D2124 FE80 41EC p[]Ai
002D2128 005C 2C78 ^,x
002D212C 0004 4EAE *[]N9
002D2130 FE80 2300 p[]#A
002D2134 002D 2162 -1b
002D2138 4E71 4CDF N9Lb
002D213C 0101 6126 []as
002D2140 2F00 4AB9 ^,J+
002D2144 002D 2162 -1b
002D2148 6714 2C78 g[]x
002D214C 0004 4EAE *[]N9
002D2150 FF7C 227A y[]'z
002D2154 000E 2C78 -[]x

4 Source (freemem.s) pc
0001
0002 * file examples/freemem2.s - Workbench version
0003
0004 * a sample Intuition program to display a window constant
0005 * the free memory figure, until it's closed
0006
0007 * this source code (C) HiSoft 1992 All Rights Reserved
0008
0009 * both source and binary are FreeWare and may be distribu
000A * so long as copyright messages are not removed
000B
000C * revision history:

```

MonAm's front panel

MonAm's Windows

As we have said, there are four different types of view through a window:

- a *register* window in which you can see the various 680x0 data and address registers, the program counter (PC), the status register (SR) and the current instruction. The values of the data and address registers are shown in hexadecimal together with some information about the locations to which the registers point.
- a *disassembly* window which shows a 680x0 disassembly of the memory that it is addressing, including any symbols that are found.
- a *memory* window which displays the contents of memory locations in hexadecimal and ASCII.
- a *source code* window. In this type of window you can view a text file which may be the source code of the program that you are debugging, assuming that this exists. You can display line numbers if you wish and, if the program that owns the source code has line number information in a HUNK.DEBUG hunk, you will be able to use this information to step through the program's source code and set breakpoints on source lines.

Up to five windows can be shown simultaneously or, by changing the width and height of the windows you, can show just two.

Each window is numbered from 1 to 5 and can display different types -of information - window 1 can be of any type, register, memory, source code or disassembly; windows 2 and 4 can be memory, disassembly or source code windows whilst windows 3 and 5 are restricted to being memory windows.

Stacking Windows

Each window also has depth - you can stack views beneath a window so that you have almost limitless flexibility in what you choose to display.

In addition you can *split* and *widen* most windows; split means to grow or to shrink the window vertically whilst widen means to do the same horizontally. These operations may hide other windows temporarily or they may uncover hidden windows.

Locking Windows

Each window may also be locked to an arbitrary expression. Thus, you can lock a memory window to a register so that it displays the contents of the memory addressed by that register. Or you might want to lock a disassembly window to the PC, which is the default condition for window 2 unless you have saved.

Each view on the window stack can be locked to a different expression although it does not make sense to lock the register window.

All the above window features will be discussed in more detail later.

The Current Window

MonAm has the concept of a current window - this is denoted by displaying its title highlighted and is the window on which any operation will take place.

The current window may be changed by pressing the Tab key to cycle between them, or by pressing the Δ key together with the window number, for example $\Delta 2$ selects window number 2, even if it is hidden currently.

NOTE: If your typing seems to be ignored in MonAm don't be alarmed; it means that another screen is active, such as that of your program. To correct this click on any part of the MonAm display. You can always tell when the MonAm display is active because the mouse pointer will be bug-shaped.

MonAm and Multi-Tasking

The Amiga® is a multi-tasking machine, and this imposes some restrictions on what MonAm can do. Having loaded a program (or task) into memory, that task is suspended. This means it is waiting, in this case for a MonAm command to let it continue.

The other state the task can be in is executing - running at the same time as MonAm. Some commands require one or other of these states to operate - for example you can only single-step a task that is suspended. If the task is running you will get the error Task must be suspended! when you try to single-step it.

MonAm can only debug one task at a time.

Symbolic Debugging

A major feature of MonAm is its ability to use symbols taken from the original program whilst debugging. MonAm uses standard AmigaDOS file HUNK_SYMBOL hunks as produced by most Amiga® programs that produce executable files, such as linkers, compilers and GenAm.

MonAm can also accept line number information from various different types of HUNK_DEBUG hunk, which enables the debugger to handle source code files that are connected with the program being debugged on a line basis. If the program to be debugged contains this line number information, you will be able to set breakpoints in its source code and even single-step it, source line by source line. Products that support this, currently, are : HighSpeed Pascal, Devpac Amiga 3, HiSoft BASIC 2 and SAS/Lattice C 5.

MonAm Requesters

MonAm makes extensive use of requesters which are similar in concept to those in Intuition programs but have several differences.

ESC to abort

Breakpoint address[,param n=*?~]

a MonAm requester

A MonAm requester displays the prompt ESC to abort above the top left corner of the box together with a prompt, normally followed by a blank line or some text to edit, with a cursor. At any time a requester may be exited by pressing Esc, or data may be entered at the cursor by normal typing. Various keys may be used to edit the text:

←, →	move the cursor left or right through the text
Shift-←, Shift-→	move the cursor to the start of the line or to the end of the line
Backspace	delete the character behind the cursor
Del	delete the character under the cursor
Alt X	delete the entire line
Esc	abandon the requester

commands available within MonAm requesters

When you have finished entering a line, press the Return key; if the line contains errors the screen will flash and the Return key will be ignored allowing correction of the data before pressing Return again.

Some MonAm requesters simply display a message together with the prompt Return; these are normally used to inform you of some form of error. The box will disappear on pressing the Return or Esc keys, whichever you find more convenient.

Command Input

MonAm is controlled by single-key commands which gives a fast user-interface, although this can take getting used to if you are familiar with a line-oriented command interface of another debugger. Users of HiSoft Devpac on other machines such as the Atari ST/TT will find many commands are identical.

In general the **A** key is the window key - when used in conjunction with other keys it acts on the *current window*. The **Ctrl** key is usually used to invoke commands connected with execution of the program that is being debugged.

Commands may be entered in either upper or lower case. Those commands whose effects are potentially disastrous require the **Ctrl** key to be pressed in addition to a command key. The keys used were chosen to be easy to remember, wherever possible. Commands take effect immediately - there is no need to press **Return** and invalid commands are simply ignored. The relevant sections of the front panel display are updated after each command so any effects can be seen immediately.

MonAm is a powerful and sometimes complex program and we realise that it is unlikely that many users will use every single command. For this reason the remainder of the MonAm manual is divided into two sections - the former is an introduction to the basic commands of the program, while the latter is a full reference section. It is possible for new users and beginners to use the debugger effectively while having only read the *Overview*; but don't be intimidated by the *Reference* section.

MonAm Overview

Starting MonAm

MonAm is invoked by typing the command

monam [Return]

or by calling it from the Devpac editor.

If you start MonAm from a CLI you can include, optionally, a program name and a command line to be passed to the program. For example,

```
monam,c:mytest examples/demo.s [Return]
```

will cause MonAm to be invoked which will load a program called mytest and pass a filename to this program.

When MonAm has loaded, the screen will look like this:

```

MonAm version 3.08 Copyright © 1997 HiSoft

1 Registers
d0 = 00000000 .... a0 = 00000000 **** **
d1 = 00000000 .... a1 = 00000000 **** **
d2 = 00000000 .... a2 = 00000000 **** **
d3 = 00000000 .... a3 = 00000000 **** **
d4 = 00000000 .... a4 = 00000000 **** **
d5 = 00000000 .... a5 = 00000000 **** **
d6 = 00000000 .... a6 = 00000000 **** **
d7 = 00000000 .... a7 = 00000000 **** **
sr = 0000          UI
pc = 00200830 ori.b # $A6, -(a0) ;FFFFFF **

2 Disassembly pc
00200830 >ori.b # $A6, -(a0)
00200834 ori.b # $74, d0
00200838 bclr d4, (a4)+
0020083A cmp2.b ??
0020083E sub1.b # $36, d0
00200842 and1.w # $28, $A.w
00200846 cmp2.b ??
0020084C move.l $20084E(pc), -(a1)
00200850 ori.b # $3F, d2
00200854 ori.b # $DF, d0
00200858 dc.w # $7CF
0020085A ori.b # 0, d0
0020085E ori.b # 0, d0
00200862 ori.b # 0, d0
00200866 ori.b # $A8, -(a0)
0020086A ori.b # $A8, -(a0)
0020086E ori.b # 0, -(a0)
00200872 ESC to abort
00200876
0020087A
0020087E
00200882
00200884
00200888

3 Memory
00000000 **** **
00000004 **** **
00000008 **** **
0000000C **** **
00000010 **** **
00000014 **** **
00000018 **** **
0000001C **** **
00000020 **** **
00000024 **** **
00000028 **** **
0000002C **** **
00000030 **** **
00000034 **** **
00000038 **** **
0000003C **** **
00000040 **** **
00000044 **** **

Executable file to load

```

The MonAm initial screen

If you started MonAm without asking for a program to be loaded, the prompt Executable file to load will appear. This gives you another chance to load a program to debug; either type the filename of the program that you want to investigate and hit Return or press Return by itself (or Esc) to quit the requester.

Should MonAm have been called from the Devpac editor, the program that you are developing will be loaded automatically or used from memory, if it was assembled there.

Debugging a Program

If you have asked MonAm to load a program to debug you may now be prompted for a command line, if you haven't already given one; enter the line you want or just press Return. MonAm will then try to load the program. If it fails, it will display

AmigaDOS error xx

You can use the *Load Program* command to try to load the program again.

Assuming the filename is valid, MonAm will load the executable file and any symbols within the file. After the file and its symbols have been loaded successfully, the message

Breakpoint

will appear; this is because MonAm places a breakpoint at the first instruction of the program and then executes it.

The most common command in MonAm is probably *single-step*, obtained by pressing Ctrl-Z (or Ctrl-Y if you find it more convenient, perhaps because you have a German keyboard). This will execute the instruction at the PC, shown in the Register window and, normally, also in the Disassembly window. After executing it the debugger re-displays the values of the windows, so you can watch the processor execute your program, step by step. Single-stepping is the best way of going through sections of code that are suspect and require deeper investigation, but it is also the slowest - you may only be interested in a section of code near the end of your program which could take a long time to reach if you have to single-step all the way. There is, of course, an answer.

A *breakpoint* is a special instruction placed into your program to stop it running and enter MonAm. There are many types of breakpoint but we will restrict ourselves to the simplest for now. A breakpoint may be set by pressing A B, then entering the address you wish to place the breakpoint. You can enter addresses in MonAm in hex (the default base), as a symbol, or as a complex expression. Examples of valid addresses are 1A2B0, prog_start, 10+mydata. If you type in an invalid address the screen will flash and allow you to correct the expression.

Having set a breakpoint you need some way of letting your program actually *run*, and Ctrl-R will do this. It will execute your program using the values of the registers displayed and starting from the PC. MonAm will be re-entered if a breakpoint has been hit, or if an exception occurs.

MonAm uses its own *screen* which is independent of your program's screen. If you use the screen gadgets at the top right of the MonAm screen you will see your current program's display. This allows you to debug programs without disturbing their output. The MonAm screen will go to the back when you run a program from within the debugger and pop to the front when a breakpoint or other exception is reached. As usual, you can drag the MonAm screen from the top to reveal your program's screen.

MonAm uses its own *windows* too, and any window may be zoomed to the full screen size by pressing `^ Z`. To return to the main display press `^ Z` again, or the Esc key. The Esc key is also the best way of getting out of anything you may have invoked by accident. The Zoom command, like all `^` commands, works on the *current window* which you can change by pressing Tab. You can dump the current window to your *printer* by pressing `^ P`.

To change the *address* from which a window displays its data, press `^ A`, then enter the new address. The locking of a window to an expression is detailed in the *Reference* section.

To quit MonAm press Ctrl-C. This returns MonAm directly to the CLI. If the task you are debugging is still running or suspended when you try and quit, you will be warned. If MonAm terminates while the task under investigation is running, the machine will crash if any exception occurs subsequently. A safer way to exit is to use the Ctrl-Q command 'to stop the task first. If you used the Debug option from the editor then Ctrl-C will always terminate your program as well as MonAm.

We hope this overview has given you a good idea of the most common features of MonAm to let you get on with the complex process of writing and debugging assembly language programs. When you feel more confident you should try and read the *Reference* section, probably best taken, like all medicine, in small doses.

MonAm Reference

This is the reference section to MonAm; it is a complete description of the features and commands of this powerful debugger.

Numeric Expressions

There are many occasions within MonAm when you will want to enter a numeric expression; perhaps to lock a window to an expression, to assign a value to a register or to set the start address of a window.

For these cases, MonAm has a full expression evaluator, based on that in GenAm, including operator precedence. The main differences are that the default base is hexadecimal (decimal may be denoted with a \ sign *not* # as in MonAm version 1), there is no concept of a type of an expression (relative or absolute), * is used only for multiplication, there are two source operators (# and ?) and there is a not-equals operator !=.

The precedence table for MonAm's operators is given below:

Precedence of Operator	Operator(s)
1	monadic minus (-) and plus (+), source operators (# and ?)
2	bitwise not (~)
3	shift left (<<) and shift right (>>)
4	bitwise And (&), Or (!) and Xor (^)
5	multiply (*) and divide (/)
6	addition (+) and subtraction (-)
7	equality (=) less than (<), greater than (>), inequality (<> and !=), less than or equals (<=), greater than or equals (>=)

Symbols may be referred to and are normally case-insensitive and significant to 32 characters although this can be changed with the MonAm Preferences command.

Registers may be referred to simply by name, such as A3 or d7 (case insensitive), but this causes a clash with certain hex numbers. To obtain such hex numbers precede them with either a leading zero or a \$ sign.

There are several reserved symbols which are case insensitive, namely CODE, SP, SR, and SSP. Both A7 and SP refer to either the user- or supervisor-stack, depending on the current value of the status register. CODE refers to the first hunk in the program. This is the same as HUNK1. The second hunk is HUNK2 and so on.

Source Operators

There are two operators which allow debugging at a source code level; these are the # and ? operators.

To use these operators, you must have a source window open which is associated with the loaded executable program. In turn, this loaded program must have been produced by a package that includes line number information in the program's HUNK.DEBUG hunk. Otherwise the # and ? operators are invalid.

The # operator takes a source line number as its argument and returns the associated memory address, within the loaded program. So, say you have the source of hello.s loaded into window 2 and the executable of hello loaded as the current program then:

```
m3 = #20
```

will set the start address of window 3 to the address of line number 20 of the hello program (assuming that window 3 is not locked to another expression).

If the line number is out of range of the source (e.g. if you ask for line number 100 when there are only 90 lines of source), the result will be the address of the first or last line of the source, accordingly. If you use the # operator when there is no line number information available, the result will be 0.

The ? operator is the reverse of #; it returns the source line number, given a memory address. If the address is out of range of the code connected with the source window, ? returns a value of 0.

If you have only one source file loaded, the use of these operators is unambiguous. However, if you have loaded two or more source files into MonAm's windows, # and ? may return unpredictable results; in this case it is best to use them when one source file is open in the current window - they will then relate to this file.

These operators allow you to perform a variety of commands on a source level such as: *Set Breakpoint*, *Run Until* and *Lock Window*. This can make the process of debugging a complex program a far simpler and less tiresome task.

Indirection

The MonAm expression evaluator also supports indirection using the { and } symbols. Indirection may be performed on a byte, word or long basis, by following the } with a period then the required size, *which defaults to long*. If the pointer is invalid, either because the memory is unreadable or even (if word or longword indirection is used) then the expression will not be valid.

For example, the expression

`{data_start+10}.w`

will return the word contents of location `data_start + 10`, assuming `data_start` is even. Indirection may be nested as you would nest ordinary parentheses.

Memory Registers

In addition there are 10 memories numbered M0 through M9, which are treated in a similar way to registers and can be assigned to using the *Register Set* command. These are available for your own use although some have special functions as described below - you can view the memory registers by zooming the register window.

The values of memories I through 5 inclusive are the current start address of the relevant window (including source code displays) and assigning to them will change the start address of the display within that window. Here's a full table of the memory registers (on the next page):

Memory Register	Contents
m0	the destination effective address of the current instruction
m1	the start address of window 1
m2	the start address of window 2
m3	the start address of window 3
m4	the start address of window 4
m5	the start address of window 5
m6	spare
m7	spare
m8	the start address of any binary file that has been loaded
m9	the end address of any binary file that has been loaded

m8 and m9 are useful if you have loaded a binary file and then want to save it out to disk again - you do not have to remember the .start and end addresses of the file, just use m8 and m9 when saving.

If window 1 has a register display in it, m1 will be meaningless but will retain any previous value.

Window Types

There are five possible windows within the MonAm display and the exact contents of these windows and how they are displayed is detailed below. The allowed types of each window are:.

Window	Allowed Type(s)
1	register, memory, disassembly, source
2	memory, disassembly, source
3	memory only
4	memory, disassembly, source
5	memory only

A window can have a number of different views attached to it; you can think of the window as a *stack*, having depth.

So, in window 2, you can view a disassembly of code, a section of memory and a portion of an ASCII file, although only one of these at a time is visible. To cycle through the different views use the *Next/Previous View* commands and to create or delete a display use the *Open View* and *Close View* commands.

Most windows can also be *split*, either vertically or horizontally so that more, or less, can be displayed within the window - this action may hide or reveal other windows and it is best to experiment with the split commands (described below) to understand how they work.

A window can be locked to an expression so that its start address is dependent on the value of that expression - see the *Lock to Expression* command below.

You can also *zoom* a window; it will then occupy the whole of MonAm's screen.

Each type of window will now be described.

Register Window

1 Registers			
d0 = 00000001	***[]	a0 = 002A8C81	0A 0000 0000 0000 0000 0000 0000 00 [].....
d1 = ABCDABCD	<I<I	a1 = ABCDABCD	** **** **** **** **** **** **** **
d2 = ABCDABCD	<I<I	a2 = ABCDABCD	** **** **** **** **** **** **** **
d3 = ABCDABCD	<I<I	a3 = ABCDABCD	** **** **** **** **** **** **** **
d4 = ABCDABCD	<I<I	a4 = ABCDABCD	** **** **** **** **** **** **** **
d5 = ABCDABCD	<I<I	a5 = ABCDABCD	** **** **** **** **** **** **** **
d6 = ABCDABCD	<I<I	a6 = ABCDABCD	** **** **** **** **** **** **** **
d7 = ABCDABCD	<I<I	a7 = 002B63A4	002A 8D80 0000 1050 002B 5288 0027 *[]*[]P*+R[]'
sr = 0008	UI N		
pc = 0026BE70	moveq #\$64, d4		

the register window

The data registers are shown in hex, together with the ASCII display of their four bytes. The address registers are also shown in hex, together with a hex display of the memory that each register is addressing. This is word-aligned or byte-aligned as necessary, with non-readable memory displayed as *. To the right of this hex display is its ASCII interpretation.

The status register is shown in hex and in flag form, additionally with U or S denoting user- or supervisor-modes.

The PC value is shown together with a disassembly of the current instruction. Where this involves one or more effective addresses these are shown in hex, together with a suitably-sized display of the memory they point to.

For example, the display

```
TST.L $12A(A3) ;00001FAE 0F01
```

signifies that the value of \$12A plus register A3 is \$1FAE, and that the word memory pointed to by this is \$0F01. A more complex example is the display

```
MOVE.W $12A(A3),-(SP) ;00001FAE 0F01 > 0002AC08 FFFF
```

The source addressing mode is as before but the destination address is \$2AC08, presently containing \$FFFF. Note that this display is always of a suitable size (MOVEM data being displayed as a quadword) and when pre-decrement addressing is used this is included in the address calculations.

Disassembly Window

```
2 Disassembly pc
0026BE70      >moveq    #$64, d4
0026BE72      moveq    #$1F, d0
0026BE74      lea      int_name(pc), a1
0026BE78      movea.l  4.w, a6
0026BE7C      jsr      -$228(a6)
0026BE80      tst.l    d0
0026BE82      beq      exit_false
0026BE86      move.l   d0, _IntuitionBase
0026BE8C      moveq    #$1F, d0
0026BE8E      lea      graf_name(pc), a1
0026BE92      movea.l  4.w, a6
```

the disassembly window

Disassembly displays show memory as disassembled instructions to the standard described below. On the left the hex address is shown, followed by any symbol, then the disassembly itself. The current value of the PC is denoted with a >, if it is visible.

You can scroll through the disassembly window as described under *Cursor Keys* below.

If the instruction has a breakpoint placed on it this is shown using square brackets ([]) afterwards, the contents of which depend on the type of breakpoint.

For stop breakpoints this will be the number of times left for this instruction to execute, for conditional breakpoints it will be a ? followed by the beginning of the conditional expression, for count breakpoints it is an = sign followed by the current count and for permanent breakpoints a * is displayed.

The exact format of the disassembled op-codes is to the Motorola standard, as GenAm accepts. All output is upper-case (except lower-case, labels) and all numeric output is in hexadecimal, except Trap numbers. Leading zeroes are suppressed and the \$ hex delimiter is not shown on numbers less than 10. Where relevant numerics are shown signed.

The only deviation from Motorola standard is the register lists shown in MOVEM instructions - in order to save display space the type of the second register in a range is abbreviated, for example

```
MOVEM.L d0-d3/a0-a2, -(sp)
```

will be disassembled as

```
MOVEM.L d0-3/a0-2,-(sp)
```

Certain library calls will be shown symbolically even if no symbol information was loaded with your program. The disassembler is intelligent and recognises a MOVE into register A6 followed by a JSR using A6 so that the call, if valid, will be displayed by name; for example

```
MOVE.L 4,A6
JSR _LVOOpenLibrary(A6)
```

The disassembler does this for the exec, graphics, dos and Intuition libraries, using the special file monam.libfile. If this file is not found in the current directory or the current libs: assignment during MonAm's initialisation then such disassembly will not occur.

Memory Window

3 Memory																																		
0026BE70	7864	701F	43FA	015C	2C78	0004	4EAE	FDD8	4A80	6700	x	d	p	[	C	6	[	\,	x	.	N	e	y	J	[	g	.							
0026BE84	014A	23C0	0026	830E	701F	43FA	0154	2C78	0004	4EAE	[	J	#	.	&	[	p	[	C	6	[	T	.	x	.	N	e	.						
0026BE98	FDD8	4A80	6700	0122	23C0	0026	8312	41F9	0026	82D0	y	0	J	[	g	.	"	#	.	&	[	A	u	.	&	[	B	.						
0026BEAC	2C79	0026	830E	4EAE	FF3A	4A80	6700	00F8	23C0	0026	.	y	.	&	[	N	e	y	.	J	[	g	.	.	.	.	.	.						
0026BEC0	8316	41F9	0026	831E	30BC	0014	317C	0014	0002	317C	[	[	A	u	.	&	[	0	0	0	4	.	[	i		.	[	.	[	i		.		
0026BED4	012C	0004	317C	0064	0006	117C	0000	0008	117C	0001	[	.	[	i		.	d	.	[	[	.	.	.	.	.	.	.	.	.	.	.	.	.	
0026BEE8	0009	217C	0026	C019	001A	217C	0002	100F	000E	217C	.	[	i		.	&	A	.	[	.	.	.	.	.	.	.	.	.	.	.	.	.	.	
0026BEFC	0000	0200	000A	317C	000F	002E	42A8	0012	42A8	0016	.	[	.	.	.	[	i		.	[	.	.	.	.	.	.	.	.	.	.	.	.	.	
0026BF10	2179	0026	8316	001E	42A8	0022	317C	0064	0026	317C	!	y	.	&	[	[	.	[	B	.	.	.	.	.	.	.	.	.	.	.	.	.	.	
0026BF24	0019	0028	317C	0280	002A	317C	00C8	002C	2C79	0026	.	[	.	.	[	i		.	[	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

the memory window

Memory displays show memory in the form of a hex address, wordformatted hex display and ASCII. Unreadable memory locations are denoted by * *. The number of bytes shown is calculated from the window width, up to a maximum of 16 bytes per line. You can scroll through the memory window as described under *Cursor Keys* below.

Source Window

```

4 Source (helloworld.s) pc
001B      moveq    #100, d4          default error return code
001C
001D      moveq    #INTUITION_REV, d0      version
001E      lea      int_name(pc), a1
001F      CALLEXEC      OpenLibrary
0020      tst.l    d0
0021      beq      exit_false          if failed then qu
0022      move.l    d0, _IntuitionBase      else save the poi
0023
0024      moveq    #GRAPHICS_REV, d0
0025      lea      graf_name(pc), a1
0026      CALLEXEC      OpenLibrary

```

the source window

The source window shows ASCII files in a similar way to a screen editor with the name of the file displayed in the title bar. The default tab setting is 8 though this can be toggled to 4 with the *Edit Window* command.

You can choose whether or not to display line numbers for the file and whether they are shown in decimal or hexadecimal. When line number information exists in the HUNK_DEBUG hunk of your program, you can use the medium level debugging features of MonAm to step through the source and set breakpoints within it, rather like you can with a source code debugger.

You can scroll through the source window as described under *Cursor Keys* below.

Cursor Keys

The cursor keys can be used on the current window, the action of which depends on the display type.

On a memory display all four cursor keys change the current address, by byte or line, while Shift ↑ and Shift ↓ move a page in either direction.

On a disassembly display ↑ and ↓ change the start address on an instruction basis, ← and → change the address on a word basis and Shift ↑ and Shift ↓ on a page basis.

On a source-code display ↑ and ↓ change the display on a line basis, and Shift ↑ and Shift ↓ on a page basis.

Window Commands

Commands that are reached through the use of the *A* (right Amiga) key are normally available at any time. Many of these commands are connected with and apply to the *current window*. The current window is denoted by having an inverse title and it can be changed by pressing Tab or *A* plus the window number.

Most window commands work in any window, zoomed or not, though when it does not make sense to do something the command is ignored.

The exceptions to the above are the *Stack*, *Unstack* and *View Stack* commands which, for ease of use, are not reached through the *A* key and do not work on a zoomed window.

A A or M

Set Address

Allows you to set the starting address of a memory, disassembly or source window (the latter only if line number information exists in the HUNK DEBUG hunk). You can use any valid expression to generate this start address e.g.

```
_main  
$C227B8  
StartProgram + 8  
PC
```

On a memory window this lets you edit memory in hexadecimal or ASCII. Hex editing can be accomplished using keys 1-9, A-F, together with the cursor keys. Pressing **Tab** switches between hex & ASCII, ASCII editing takes each keypress and writes it to memory. The cursor keys can be used to move about memory. To leave edit mode press the Esc key.

On a register display using this command is the same as using *A R*, *Register Set*, described shortly.

Within a source window this command toggles the tab setting between 4 and 8.

You cannot edit a disassembly window.

This command works on source windows and allows you to choose the line that will appear at the top of the window. If you select a line that is beyond the end of the file, the last line will be shown at the top of the window.

This allows source (with line number information), disassembly and memory windows to be locked to a particular expression. After any exception the start address of the display is re-calculated, depending on the locked expression. Each stacked view within a window can have its own lock.

MonAm will ignore you if you try to lock a source window that refers to a program that does not have line number information attached to it.

If you try to lock a source window to an expression that lies outside the address range of the source file you will be ignored. This, in fact, is very useful; it means that if you have a stack of source windows (see below for details of stacking windows) which make up the executable that you are debugging and you lock each display to the PC, you will be able to trace the path of the program through each source file.

If an instruction in the top view calls a subroutine in the stack, the top view will not change but, if you then view the relevant stacked view, it will change to show you the called subroutine.

To unlock, simply enter a blank string.

You can lock one window to another window by using the memory registers such as M2. You can even lock a window to the indirection of its own memory register (e.g. {m2}) which might be useful to step through a linked list (in conjunction with the Esc key to update the window each time).

A P

Print Window

Dumps the current window contents onto the printer or to a file. This command can be aborted by pressing Esc. You can choose the printer device and the filename using the Preferences command.

A S

Split Window

Splits a window vertically i.e. makes it taller or shorter depending on its current state; this may hide or uncover another window. You would normally use this to set up the display as you like it and then save the set-up with the *Save Preferences* command. It can be useful at any time, though, if you would like to see more information in a window or you need another window; this command has no effect on window 1.

Note that as an alternative using *A* 1-5 will split the windows in such a manner as to make the selected window visible. Also beware that unlike MonAm2 splitting a window will not automatically select the newly opened window.

A T

Type

This command works on windows 1, 2 and 4; it changes the type of the display between register (for window 1), disassembly, memory and source (if a source file has been loaded into the window).

Splits a window horizontally i.e. makes it wider or narrower depending on its current state; this may hide or uncover another window. You would normally use this to set up the displa as you like it and then save the set-up with the *Save Preferences* command. It can be useful at any time, though, if you would like to see more information in a window or you need another window.

This command has no effect on window 1.

This zooms the current window to be full size. Other A commands are still available and normal size can be achieved by pressing Esc or A Z again.

Zooming a register window shows some extra information (which depends on the processor type) and the memory registers (m0-m9):

```

MonAm version 3.08 Copyright © 1997 HiSoft
1 Registers
d0 = 00000001  ***[] a0 = 002A8C81  0A 0000 0000 0000 0000 0000 0000 00  [].....
d1 = ABCDABCD  <I<I  a1 = ABCDABCD  ** **** **** **** **** **** **** **
d2 = ABCDABCD  <I<I  a2 = ABCDABCD  ** **** **** **** **** **** **** **
d3 = ABCDABCD  <I<I  a3 = ABCDABCD  ** **** **** **** **** **** **** **
d4 = ABCDABCD  <I<I  a4 = ABCDABCD  ** **** **** **** **** **** **** **
d5 = ABCDABCD  <I<I  a5 = ABCDABCD  ** **** **** **** **** **** **** **
d6 = ABCDABCD  <I<I  a6 = ABCDABCD  ** **** **** **** **** **** **** **
d7 = ABCDABCD  <I<I  a7 = 002B63A4  002A 8D80 0000 1050 002B 5288 0027  *[]*[]P*+R[]*
sr = 0008      UI N
pc = 0026BE70  moveq #$64, d4

ssp = 002022A4  msp = 00000000  sfc = 00  cacr = 0001
vbr = 00000000  isp = 0020229C  dfc = 00  caar = 00000000

fpcr = 0000  fpr = 00000000  fpiar = 00000000
fp0 = 0000 00000000 00000000  0.0000000000000000e+0000
fp1 = 0000 00000000 00000000  0.0000000000000000e+0000
fp2 = 0000 00000000 00000000  0.0000000000000000e+0000
fp3 = 0000 00000000 00000000  0.0000000000000000e+0000
fp4 = 0000 00000000 00000000  0.0000000000000000e+0000
fp5 = 0000 00000000 00000000  0.0000000000000000e+0000
fp6 = 0000 00000000 00000000  0.0000000000000000e+0000
fp7 = 0000 00000000 00000000  0.0000000000000000e+0000

m0 = FFFFFFFF  ** **** **** **** **** **** **** **
m1 = 00000000  **** **** **** **** **** **** **** **
m2 = 0026BE70  7864 701F 43FA 015C 2C78 0004 4EAE FDD8  xdp[]C[]\, x*[]N@y@
m3 = 0026BE70  7864 701F 43FA 015C 2C78 0004 4EAE FDD8  xdp[]C[]\, x*[]N@y@
m4 = 002B65DB  09 6D6F 7665 7109 2331 3030 2C64 3409 09  []moveq[]#100, d4[]
m5 = 00000000  **** **** **** **** **** **** **** **
m6 = 00000000  **** **** **** **** **** **** **** **
m7 = 00000000  **** **** **** **** **** **** **** **
m8 = 00000000  **** **** **** **** **** **** **** **
m9 = 00000000  **** **** **** **** **** **** **** **

```

the zoomed register display (on a 68020 machine)

NOTE: A zoomed window behaves differently from a normal window in that, as you scroll through it, it does not update the associated memory register (m1 to m5). Also, if you change the value of the memory register while in a zoomed window, the start address of the display will not change. Think of a zoomed window as only temporary.

Shift-.

Open View

Creates a new view on the current window and numbers it accordingly. The type of the new view will be the same as the previous one if this is possible.

The display will be numbered xa, xb, xc, xd etc. where x is the number of the window e.g. if you stack a new display on window 2, it will be numbered 2b with the original display being numbered 2a. Remember, though, that there is only one memory register per window, but you can lock each display to a different expression. This gives a tremendous amount of flexibility.

This command does not work on a zoomed window.

The associated memory register is bound to the top view only, although all locks on all views are re-calculated where necessary.

Shift-,

Close View

Removes the visible display from the current window's display list, unless there is only one display attached to this window, in which case the command does nothing. If you close a view on a source window, the source file will be un-loaded from memory and a disassembly window will replace the closed source view.

All other displays attached to this window will be re-numbered if necessary i.e. if you remove display 2c from (2a, 2b, 2c, 2d), 2d will be re-numbered to be 2c.

This command does not work on a zoomed window.

These two commands allow you to cycle through views that have been stacked onto a window. Pressing . (full stop or period) cycles forward through the available displays whilst , (comma) cycles backwards. Both will roll round in a loop.

For example, say you have 3 displays stacked on window 4 (4a Source, 4b Memory and 4c Disassembly) and you are currently displaying 4b Memory. Press . and 4c Disassembly will appear, press . again and you will see 4a Source.

These commands do not work on a zoomed window.

Esc

Pressing Esc will update all the window displays, if necessary and recalculate the addresses to which any windows and views are locked.

This can be very useful in many cases; for example say you have window 3 locked to {m5} (the address pointed to by window 5) and you then scroll through window 5. Normally this will not update window 3. However, all you have to do is to press Esc when you want to update window 3 (and all the other windows).

You can press the Esc key while your program is running to see how the machine state is changing (assuming that the MonAm screen is at the front).

Other Commands

All  (right Amiga) commands (like the window commands described above) are available for use at any time whilst you are using MonAm. There are a few other such commands that are not related to the current window:

B

Set Breakpoint

Allows the setting of any type of breakpoint, described later under *Breakpoints*.

This prompts for an expression and displays its value in hexadecimal, decimal and as a symbol if relevant.

```
ESC to abort

Enter expression
m2

=$0026BE70, 2539120█
```

example of Show Other

Allows any register to be set to a value, by specifying the register, an equals sign and its new value. It can also be used to set the value of the memory registers. For example the line

a3=a2+4

sets register A3 to be A2 plus 4 whereas:

m3=m2

will set the value of the window 3 register to be the same as the window 2 register. All windows will then be re-drawn, which may cause a display that you did not expect if, say, the display in window 3 is locked to an expression.

You can also use this to set the start address of a window when in zoom mode so that, on exit from zoom mode, the relevant window starts at the required address.

Screen Switching

MonAm uses its own Screen display and will always make itself the front and active window whenever an exception (including breakpoints) occurs.

This will put the MonAm screen to the back, normally showing your program's screen. Pressing any key will return the MonAm screen (so long as you have not activated any other window).

Breakpoints

Breakpoints allow you to stop the execution of your program at specified points within it. MonAm allows up to eight simultaneous breakpoints, each of which may be one of five types. When a breakpoint is hit MonAm is entered and it then decides whether to halt execution of your program (when' it will enter the front panel display) or continue; this decision is based on the type of the breakpoint and the state of your program's variables.

Simple Breakpoint [1]

These are one-off breakpoints which, when executed, are cleared and cause MonAm to be entered.

Stop Breakpoint [n]

These are breakpoints that cause program execution to stop after the breakpointed instruction has been executed a specified number of times. In fact a simple breakpoint is really a stop breakpoint with a count of one.

Count Breakpoint [=]

Merely counters; each time such a breakpoint is reached a counter associated with it is incremented, and the program will resume. These breakpoints are more like monitors - they never cause a program to stop and are useful for profiling.

Permanent Breakpoint [*]

These are similar to simple breakpoints except that they are never cleared - every time execution reaches a permanent breakpoint MonAm will be entered.

Conditional Breakpoint [?]

The most powerful type of breakpoint; these allow program execution to stop at a particular address only if an arbitrarily complex set of conditions apply. Each conditional breakpoint has associated with it an expression (conforming to the rules already described). Every time the breakpoint is reached this expression is evaluated, and if it is non-zero (i.e. true) then the program will be stopped, otherwise the program will continue.

A B

Set Breakpoint

This is a window command allowing the setting or clearing of breakpoints at any time. The line entered should be one of the following forms, depending on the type of breakpoint required:

`<address>`

will set a simple breakpoint.

`<address> , <expression>`

will set a stop breakpoint at the given address, which will execute `<expression>` times. The expression is evaluated before the program is executed.

`<address> , =`

will set a count breakpoint. The initial value of the count will be zero.

`<address> , *`

will set a permanent breakpoint.

`<address> , ?<expression>`

will set a conditional breakpoint, using the given expression.

`<address> , -`

will clear any breakpoint at the given address.

Breakpoints cannot be set on addresses which are odd, unreadable, or within ROM.

Every time a breakpoint is reached, regardless of whether the program is interrupted or resumed, the program state is remembered in the History buffer, described below

Help *Show Help and Breakpoints*

This displays the current breakpoints, task status, its segment list (showing where your program is), free memory and the system memory list.

UK A500 1.2 users (who cannot use the Help key) can also obtain this command by pressing A ii.

Ctrl-B *Simple Breakpoint*

Included mainly for compatibility with MonAm 1, this sets a simple breakpoint at the start address of the current window, so long as it contains a disassembly display. If a breakpoint is already there then it will be cleared.

U *Run Until*

This prompts for an address and a breakpoint specifier (1, n, =, */ or ?). The chosen type of breakpoint is then placed at the given address and program execution resumed.

Ctrl-K *Kill Breakpoints*

Clears all set breakpoints. This is also done automatically when you quit MonAm with a task still running.

A command that places a simple breakpoint at the instruction *after* the instruction at the PC and resumes execution from the PC. This is particularly useful for DBF-type loops if you don't want to go through the loop, but just want to see the result after the loop is finished .

This is a command to stop your task while it is executing. It does this by forcing the Trace bit to be set, so you will get a Trace exception. While this does work, be very careful if you stop a task in the middle of some AmigaDOS ROM routines, particularly signal handling and message passing.

NOTE: The above command accesses memory fields that are not guaranteed to remain the same for different versions of the Amiga® operating system.

NOTE: This command was Ctrl-S in MonAm version 1

History

MonAm has a *history buffer* in which the machine status is remembered for later investigation.

The most common way of entering data into the history buffer is when you single-step, but in addition every breakpoint reached and every exception caused enters the machine state into the buffer. The various forms of the Run command also cause entries to be made into this buffer.

NOTE: The history buffer has room for five entries - when it fills the oldest entry is removed to make room for the newest entry.

This opens a large window displaying the contents of the history buffer. All register values are shown including the PC as well as a disassembly of the next instruction to be executed.

NOTE: If a disassembly in the History display includes an instruction which has a breakpoint placed on it, the [] s will show the *current* values for that breakpoint, not the values at the time of the entry into the history buffer.

Quitting MonAm

Ctrl-C or *A* Q

Exit MonAm

This exits MonAm, returning control to whatever task invoked MonAm. All breakpoints are killed although, if the task you are debugging is still running or suspended when you try and quit, you will be warned. If MonAm terminates while the task under investigation is running, the machine will crash should any exception occurs subsequently. A safer way is to use the Ctrl-Q command to stop the task first.

If the Debug option has been used from the GenAm editor then MonAm will terminate automatically when the program it is debugging has terminated.

Ctrl-Q

Quit a program

This is a way of forcing the task being debugged to quit. This can be hazardous to use, and should only be done as a last resort. If your program is terminated in this way it will not clean up, and thus not de-allocate any memory it was using or close windows etc.

NOTE: The above command accesses memory fields that are not guaranteed to remain the same for different versions of the Amiga® operating system.

Loading & Saving

Ctrl-L

Load Program

This will prompt for a filename and a command line and will attempt to load the file ready for execution. If MonAm has already loaded a program it is not possible to load another until the former has terminated.

The file to be loaded must be an executable file. Use the *Load Binary File* command if you wish to edit other file types.

NOTE: **This command is not available if MonAm has been invoked using *Debug* from the editor.**

B

Load Binary File

This will prompt for a filename and an optional load address (separated by a comma) and will then load the file where specified. If no load address is given then memory will be allocated from the system. M8 will be set to the start address of the loaded file and M9 to the end address.

NOTE: **This is a change from previous versions of MonAm, where M0 and M1 were set to the start and end addresses of the loaded file.**

S

Save Binary File

This will prompt for a filename, a start address and an (inclusive) end address. To re-save a file recently loaded with the *Load Binary File* command

<filename>,M8,M9

may be specified, assuming of course that M0 and M1 have not been re-assigned.

This powerful command allows an ASCII file, normally of source code, to be loaded and viewed within MonAm. This can be loaded into window 2 or window 4. If the loaded program has line number information relevant to this source file; you will be able to use line number operators on this display to step through the source code, set breakpoints within it etc.

A new view on this window will be opened if the window already contains an ASCII file, otherwise the text will replace the current window. You can unload a source window using the Close View command.

The source window will be locked automatically to the PC.

Memory for source code displays is taken from the system so sufficient free memory must be available.

Executing Programs

Ctrl-R

Return to program /Run

This runs the current program with the given register values at full speed and is the normal way to resume execution after entry via a breakpoint or an exception.

Ctrl-Z

Single-Step

Single-steps the instruction at the PC with the current register values. Single-stepping a Trap, Line-A or Line-F opcode will, by default, be treated as a single instruction.

Ctrl-Y

Single-Step

Identical to Ctrl-Z above but included for the convenience of users of German keyboards.

Ctrl-T

Trace Instruction

This interprets the instruction at the PC using the displayed register values. It is similar to Ctrl-Z but obeys BSRs, JSRs, Traps, Line-A and Line-F calls as if one instruction, re-entering the debugger on return from them to save stepping all the way through the routine or trap. It works on instructions in ROM or RAM.

Ctrl-S

Skip Instruction

Ctrl-S increments the PC register by the size of the current instruction thus causing it to be skipped. Use this instead of Ctrl-Z when you know that this instruction is going to do something it shouldn't.

R

Run (various)

This is a general Run command and prompts for the type of execution, selected by pressing a key.

Run G Go

This is identical to Ctrl-R and resumes the program at full speed.

Run I Instruction

This executes the entered number of instructions remembering information in the History buffer and then returning to MonAm.

Traps are treated as single-instructions.

Searching Memory

G

search memory (Get a sequence)

You will see the prompt Search for B/W/L/T/I?, standing for Bytes, Words, Longs, Text and Instructions.

If you select B, W or L you will then be prompted to enter the sequence of numbers you wish to search for, each separated by commas. MonAm is not fussy about word-alignment when searching, so it can find longs on odd boundaries, for example.

If you select T you may search for any given text string, for which you will be prompted. The search will be case-dependent or caseindependent, as you have chosen.

If you select I you can search for part or all of the mnemonic of an instruction, for example if you searched for \$14 (A you would find an instruction like `MOVE.L D2,$14(A0)`). The case of the string you enter is un-important unless you have chosen it to be so, but you should bear in mind the format that the disassembler produces, e.g. always use hex numbers, refer to A7 rather than SP and so on.

Having selected the search type and parameters, the search begins, control passing to the Next command, described below.

Searching Source-Code Windows

If the G command is used on a source-code window the T subcommand is automatically chosen and if the text is found the window will re-display the line containing it.

N

find Next

N can be used after the G command to find subsequent occurrences of the search data. With the B, W, L and T options you will always find at least one occurrence, which will be in the buffer within MonAm that is used for storing the sequence. With the T option you may also find a copy in the system keyboard buffer. With these options, the ESC key is tested every 64k bytes and can be used to stop the search. With the I option, which is very much slower, the ESC key is tested every 2 bytes.

The search will start just past the start address of the current window (except register windows) and if an occurrence is found redisplay the window at the given address.

The search area of memory goes from 0 to the end of chip memory, then \$F80000 to \$FFFFFF (the ROM) then any additional RAM.

Miscellaneous

Ctrl-P

Preferences

This permits control over various options within MonAm. The first three require Y / N answers, pressing Esc aborts or Return leaves them alone.

```
MonAm version 3.08 Copyright © 1997 HiSoft
1 Registers
d0 = 00000001  <I<I  a0 = 002A8C81  0A 0000 0000 0000 0000 0000 0000 00  [].....
d1 = ABCDABCD  <I<I  a1 = ABCDABCD  ** **** **
d2 = ABCDABCD  <I<I  a2 = ABCDABCD  ** **** **
d3 = ABCDABCD  <I<I  a3 = ABCDABCD  ** **** **
d4 = ABCDABCD  <I<I  a4 = ABCDABCD  ** **** **
d5 = ABCDABCD  <I<I  a5 = ABCDABCD  ** **** **
d6 = ABCDABCD  <I<I  a6 = ABCDABCD  ** **** **
d7 = ABCDABCD  <I<I  a7 = 002B63A4  002A 8D80 0000 1050 002B 5288 0027  .*[]..[]P.+R[].'
sr = 0008      UI N
pc = 0026BE70  moveq #$64,d4
2 Disassembly pc
0026BE70      >moveq  #$64,d4
0026BE72      moveq  #$1F,d0
0026BE74      lea     int_name(pc),a1
0026BE78      movea.l 4,w,a6
0026BE7C      jsr     -$228(a6)
0026BE80      tst.l   d0
0026BE82      beq     exit_false
0026BE86      move.l  d0,intuitionBase
0026BE8C      ESC to abort  iF,d0
0026BE8E
0026BE92
3 Memory
0026BE70  7864 701F xdp[]
0026BE74  43FA 015C Cg[]\
0026BE78  2C78 0004 ,x*[]
0026BE7C  4EAE FDD8 New[]
0026BE80  4A80 6700 Jlg[]
0026BE84  014A 23C0 []#[]
0026BE88  0026 830E &[][]
0026BE8C  701F 43FA p[]Cg[]
0026BE90  0154 2C78 []T,x
0004 4EAE []N[]
FDD8 4A80 ýJ[]
6700 0122 g*[]
23C0 0026 #A.&[]
8312 41F9 [][]A[]
0026 82D0 &[]E[]
2C79 0026 ,y.&[]
830E 4EAE []N[]
FF3A 4A80 ýJ[]
6700 00F8 g*.*[]
23C0 0026 #A.&[]
8316 41F9 [][]A[]
0026 831E &[][]
30BC 0014 0*.*[]
317C 0014 1[]|[]
4 Source (hello)
001B      mo
001C
001D      mo
001E      le
001F      CA
0020      ts
0021      be
0022      mo
0023
0024      no
0025      le
0026      CA
PREFERENCES
Auto-load source file Y/N? Y
Source window line numbers H/D/N? H
Automatic ' ' or '0' prefix Y/N? Y
Case insensitive symbols Y/N? Y
Symbol significance
\32
Show relative offset symbols Y/N? Y
Show ZAn in disassembly Y/N? N
Interlace Y/N/D? D
Printer device name
Save preferences Y/N? []
```

the Preferences display

Auto-load source file

when switched to Yes, upon loading a program, MonAm will attempt to load the first source file associated with the program. This will only occur if the executable file contains line number debugging information and the source file can be found in the current directory. The new source file window will then be locked to the Program Counter in order to track program flow. This is of most use when debugging a program generated from a single source file.

Source window line numbers

Affects whether line numbers are shown for all debugger source windows. You may select No line numbers, Decimal numbers or Hex numbers. Hexadecimal is often the preferred setting because by default, MonAm treats all numbers as hex. Decimal line numbers, used with the # operator for example, require a prefix of backslash.

Automatic '_' or '@' prefix

This is provided mainly for the convenience of C compiler users. With it enabled, MonAm will automatically add a leading underscore or @ character to the appropriate symbols. However, symbols without the leading character will still take precedence.

Case insensitive symbols

MonAm version 3 defaults to using *case insensitive* symbols, i.e. upper and lower case characters are not distinguished between. Selecting No will mean that you must match the case of each symbol character exactly as with previous versions of MonAm.

Symbol significance

This prompts for the significant length of symbols, which is normally 32 but may be reduced to as low as 8 or increased if required. Although reducing this can save some typing, using too low a value can make some symbols impossible to select.

Show relative offset symbols

This option defaults to YL-s and affects the disassembly of *address register indirect with offset* addressing modes, i.e. `xxx(An)`. With the option on, the current value of the given address register is added to the offset then searched for in the symbol table. If found it is disassembled as symbol (An). This option is very useful for certain styles of assembly language programming as well as high level languages which use a base register as a major offset, such as SAS/C which uses A4 as a pointer to the merged data section.

Show ZAn in disassembly

Is normally switched off but advanced programmers may wish to enable the display of the normally hidden Z registers used by some 680x0 instructions.

Interlace

Allows you to select a double height interlaced screen for MonAm. This option will take effect after preferences have been saved and MonAm restarted. MonAm will normally replicate the Workbench screens format.

Printer device name

This lets you set the device that MonAm uses for its printer commands. The default is PRT:, the system printer device

configured through Preferences You may specify an AmigaDOS filename in order to re-direct printing to disk.

Save preferences

Reply Y to this command to save your current preferences to the file MonAm.prefs in the current directory. When MonAm loads it will read your preferences from this file. MonAm.prefs is firstly searched for in the current directory, then in the ENV:devpac directory, in a similar way to the editor preferences file.

I

Intelligent Copy

This copies a block of memory to another area. The addresses should be entered in the form

<start>,<inclusive_end>,<destination>

The copy is intelligent in that the block of memory may be copied to a location which overlaps its previous location.

NOTE: No checks at all are made on the validity of the move, copying to non-existent areas of memory is likely to crash MonAm and corrupting system areas may well crash the machine.

L

List Labels

This opens up a large window and displays all loaded symbols. Any key displays the next page, pressing Esc aborts. The symbols will be displayed in the order they were found on the disk (or in memory if using the Debug option from the editor).

This command can only be used if you are debugging a task which had a symbol table loaded with it. What it does is de-allocate the memory used for storing the symbols, freeing it for the system to use. This can be very useful if memory is tight while debugging a larger program, as you can load it, together with symbols, set a breakpoint at a symbolic address, then lose the labels before letting it run. Of course once you hit your breakpoint you won't have any symbols.

NOTE: This command was Ctrl-I-L in MonAm version 1

This fills a section of memory with a particular byte. The range should be entered in the form

<start>,<inclusive-end>,<fillbyte>

The warning described previously about no checks applies equally to this command.

This command allows the disassembly of an area of memory to printer or disk, complete with original labels and, optionally, an automatic list of labels' created by MonAm, based on cross

references. The first line should be entered as

<start_address>,<end_address>

The next line prompts for the area of memory used to build the cross-reference list, which should be left blank if no automatic labels are required else should be of the form

<buffer_start>,<buffer_end>

Next is the prompt for data areas which will be disassembled as DC instructions, of the form

<data-start>,<data-end>[,<size>]

The optional size field should be B, W or L, defaulting to L, determining the size of the data. When all data areas have been defined, a blank line should be entered.

Finally a filename prompt will appear; if this is blank all output will be to the printer, else it will be assumed to be a disk file.

If automatic labels were specified there may be a delay at this point while the table is generated. Automatic labels are of the form Lxxxxx where xxxxx is the actual hex address.

Printer Output

This is of the form of an 8 digit hex number, then up to 10 words of hex data, 12 characters of any symbol, then the disassembly itself. Printer output may be aborted by pressing Esc.

Disk Output

This is in a form directly loadable by GenAm, consisting of any symbol, a tab, then the disassembly itself, with a tab separating any operand from the opcode. If you are disassembling an area of memory without loaded symbols then the X R E F option should be used else no symbols will appear at all in the output file. Pressing Esc or a disk error will abort the disassembly.

M Modify Address

Included for compatibility with MonAm 1, equivalent to `A A`.

0 Show Other Bases

Included for compatibility with MonAm 1, equivalent to `A O`.

D Change Drive & Directory

This allows the current drive and sub-directory to be changed.

Command Summary

Window Commands

A A	Set Address
A B	Set Breakpoint
A E	Edit View
A G	Goto Source Line
A L	Lock to Expression
A P	Print Window
A R	Register Set
A S	Split Window
A T	Change Type
A W	Widen Window
A Z	Zoom Window
Shift-.	Open View
Shift-,	Close View
. and ,	Next/Previous View
Esc	Update all Windows
Tab	Activate next window
A -1-5	Activate window 1 to 5

Breakpoints

Ctrl-A	Breakpoint After
Ctrl-B	Simple Breakpoint
Ctrl-K	Kill Breakpoints
Ctrl-X	Stop Executing
A B	Set Breakpoint
U	Run Until
Help	Show Help and Breakpoints

Loading and Saving

Ctrl-L	Load Program
A	Load ASCII File
B	Load Binary File
S	Save Binary File

Executing Programs

Ctrl-R	Return to program / Run
Ctrl-S	Skip Instruction
Ctrl-T	Trace Instruction
Ctrl-Y	Single-Step
Ctrl-Z	Single-Step
R	Run (various)

Searching Memory

G	Search Memory (Get a sequence)
N	Find Next

Miscellaneous

Ctrl-C or $\mathcal{A}$ Q	Exit MonAm
Ctrl-Q	Quit a program
Ctrl-P	Preferences
Ctrl-U	Unload symbols
$\mathcal{A}$ O or O	Show Other Bases
D	Change Drive & Directory
H	Show History Buffer
I	Intelligent Copy
L	List Labels
M	Modify Address
P	Disassemble to Printer/Disk
V	View other Screen
W	Fill Memory With

Debugging Stratagem

Restrictions

As it runs as a process MonAm relies on the `exec`, intuition and graphics libraries. If your program starts destroying memory to which it has no right it is possible for it to corrupt fatally something the system needs so that when MonAm is entered after an exception the machine will crash. Fortunately this type of error is rare, usually address errors occur before programs start destroying memory.

When a program is invoked from MonAm it is set up to look as if it has been run from the CLI, not the Workbench.

MonAm cannot single-step or breakpoint any code when executing in Supervisor mode. This is because the `exec` exception handler checks for an exception in supervisor mode, and will put up a Guru alert if this is the case. If not it will enter MonAm and work normally.

If your program creates another program or task you cannot use MonAm to breakpoint it or single-step. MonAm can only debug the program that was specified when it initially loaded.

Don't try and run the standard system programs from within MonAm, such as `dir`. These programs rely on undocumented registers (particularly A5) and memory areas which MonAm cannot emulate.

Owing to a hardware feature, a word- or longword-access on odd memory locations 1 to 7 inclusive will cause a complete machine crash. There seems to be nothing we can do to prevent this.

Bug Hunting

There are probably as many strategies for finding bugs as there are programmers; there is really no substitute for learning the hard way, by experience. However, here are some hints which we have learnt, the hard way!

Firstly, a very good way of finding bugs is to look at the source code and think. The disadvantage of reaching first for the debugger, then second for the source code, is that it gets you into bad habits. You may switch to a machine or programming environment that does not offer low-level debugging, or at least not one as powerful you are used to.

If a program fails in a very detectable way, such as causing an exception, debugging is normally easier than if, say, a program sometimes doesn't quite work exactly as it should.

Many bugs are caused by a particular memory location being stepped on. Where the offending memory location is detectable, by producing a bus error, for example, a conditional breakpoint placed at one or more main subroutines can help greatly. For example, suppose the global variable `main_ptr` is somehow becoming odd during execution, the conditional expression could be set up as

```
{ main_ptr } &1
```

Count breakpoints are a good way of tracking down bugs before they occur. For example, suppose a particular subroutine is known to eventually fail but you cannot see why, then you should set a count breakpoint on it, then let the program run. At the point where the program stops, because of an exception say, look at the value of the count breakpoint (using `Help`). Terminate the program, re-load it, then set a stop breakpoint on the subroutine for that particular value or one before it. Let it run, then you can follow through the sub-routine on the very call that is fails on, to try and work out why.

Good luck!

Exception Analysis

When an unexpected exception occurs, it's very useful to be able to work out where and why it occurred and, possibly, to resume execution. Some of the most common exception types are listed below with their possible causes.

Bus Error

If the PC is in some non-existent area of memory then look at the relevant stack to try and find a return address to give a clue as to the cause, probably an unbalanced stack (i.e. some data was placed on the stack and never retrieved, causing the program to RTS to an incorrect location).

If the PC is in a correct area of your program then the bus error must have been caused by a memory access to non-existent or protected memory. Recovering from bus errors and resuming execution is generally difficult.

Address Error

If the PC is somewhere strange then the method above should be used, otherwise the error must have been caused by a program access to an odd address. Correcting a register value may be enough to resume execution, at least temporarily.

Note well that 68020 processors and upwards do not consider this an error and can quite happily read words from odd addresses, albeit at reduced efficiency. This is often the cause of programs working on a 680x0 which crash on machines with a 68000. However, such processors will cause an address error if the PC becomes odd.

illegal Instruction

If the PC is in very low memory, below around \$30, it is probable that it was caused by a 'jump to location 0. If you use MonAm to look here you will normally various ORI instructions (really longword pointers) and eventually an illegal instruction.

Privilege Violation

This is caused by executing a privileged instruction in user mode, normally meaning your program has gone horribly wrong. Bumping the PC past the offending instruction is unlikely to be much help in resuming the program.

If you really require to execute a privileged instruction you may call it via the Exec library's Supervisor routine.

Divide by Zero

Signifies that your program has attempted to divide another value by zero. This is normally due to an error in some previous calculation.

Trap instructions

These range of instructions are shared between all tasks running on the Amiga. They are unused by the operating system and are available for individual programs. As such, they will not cause an exception within MonAm but may well cause a machine crash if incorrectly used.

In order to make use of a specific numbered trap it is important to first successfully call Exec library's AllocTrap routine. This takes care of trap arbitration, ensuring that no other program is currently using that trap. Your program may then write to the trap vector and proceed to use TRAP instructions upon it, remembering to call FreeTrap during program termination. No attempt to use a trap which has not been previously allocated should be made.

Floating Point Exceptions

Such exceptions are caused by a 68881 or 68882 maths coprocessor after some error in calculation or protocol is detected. They differ from most exceptions in that they will often be caused some time after the error was actually detected, typically at the time of the next floating point instruction. This is due to the fact that an FPU instruction merely initiates the sequence of actions, the processor itself will continue to execute the instruction concurrently with further CPU instructions.

Chapter 5

Blink

The Linker

Blink is the de-facto standard AmigaDOS linker and is used directly from the command line. The linker command line specifies which files are to be linked together and in what order. Note that the order of linking is significant as this allows a symbol defined in a module linked earlier to override one in a later module; this is often useful when replacing standard library routines with your own custom versions.

A simple Blink command line

To link two object file together the command line used could be:

```
BLINK FROM first.o+second.o
```

this will produce an executable program (assuming no errors occur) named *second*; note that the name of the executable is taken from the *second* named file in the link sequence (or the first if only one is available).

Concepts

Blink provides several unusual features whilst linking, this allows more flexibility when initially writing your program leaving many of the decisions up to the linker.

ALVs

When Blink is collecting all of the CODE type sections together, if any are more than 32K apart and a 16-bit PC relative access is attempted, rather than simply fail with an out-of-range error message, Blink redirects the access to a JMP to the same location. This jump is known as an *automatic link vector* or *ALV*. Note that this may cause problems if you attempt to access data using PCrelative mode, although this is not recommended anyway since on the 68020, 68030 and 68040 there are separate code and data caches which can cause consistency problems.

Near DATA/BSS

Blink supports a 64K near data section which can be accessed via a global base register (traditionally A4). This section is formed from all sections which are named `_MERGED` (as described in the assembler section) and then several variables are created by the linker to allow the initial base of this to be set up. This is discussed later under the *Reserved symbols* section.

Directives

The Blink directives allow the input files and the format of the output file to be specified.

Input directives

The input directives allow the names of the object files to be linked to be passed to the linker. The linker works by collecting all sections which have identical names into a single output section.

FROM files Specifies the object files that are the input files for the linker. If the first item on the command line is a filename then the FROM keyword is optional and may be omitted. FROM may be used more than once with the files for each FROM adding to the list of files to be linked. Note that ROOT is a synonym for FROM.

To specify more than one file in a single FROM statement they may either be listed after it separated by spaces or +, e.g.

FROM a.o+b.o

LIB files Specifies the files to be scanned as libraries. Only modules within the library which contain symbols which are referenced will be included in the final object module. Note that LIBRARY is a synonym for LIB. The same syntax used for specifying multiple FROM files may be used for multiple libraries.

Output directives

The output directives control the format and type of the final file created by the linker when a link has been completed successfully.

ADDSYM This causes Blink to emit symbols for all exported symbols in the input object files regardless of whether the input object file was compiled with one of the debug options.

CHIP Forces *all* hunks are to be placed in 'Chip' memory regardless of the input object hunk specifications.

FAST Forces *all* hunks are to be placed in 'Fast' memory regardless of the input object hunk specifications.

MAXHUNK n Limits the maximum size hunk that Blink will create when coalescing hunks. This can be used to control fragmentation of memory. The default is no limit on hunk size.

NODEBUG	Suppresses any symbol table information or symbolic debug information in the final object file. Note that ND is a synonym for NODEBUG.
PRELINK	Causes Blink to output an object module with references and definitions still intact so that it can be linked later on to produce a final executable file. This is designed for development of large projects where the programmer is only changing a single source module. Note that a prelinked object file cannot have ALVs inserted into it and so Blink may be unable to satisfy all 16 bit PC-relative references when linking with the prelinked file.
SMALLCODE	Forces <i>all</i> CODE hunks to be coalesced into a single hunk. Note that SC .is a synonym for SMALLCODE.
SMALLDATA	Forces all DATA and BSS sections to be coalesced into a single hunk. This is useful for combining all data hunks from a program into a single hunk, decreasing load time but producing larger hunks that are difficult to scatter load. Note that SD is a synonym for SMALLDATA .
TO file	Specifies the name of the output file which is to be created.

Pre-linking

Pre-linking is similar to a normal link, however instead of producing an executable file, it coalesces only identically typed and named sections into output sections. If a section is unnamed then it is merged with the first named section of the same type. Note that the special name `__MERGED` is considered a typemodifier and hence only sections named `__MERGED` will be coalesced with `__MERGED` sections.

When pre-linking ALVs are applied according to the normal rules, i.e ALVs will be generated for out of range branches within a section, and *all* cross-output-section references.

During pre-linking variables will often have undefined values (since the modules in which these are defined are to be linked later) and so all of these variables are reported. Note that this means that an error has technically occurred and the return code from Blink will be non-zero. This is likely to be important to users of make type utilities.

Map files

A map file is a file describing the order and location of files and variables processed by the linker written to a normal file for perusal by the user. These files provide a large number of options for the programmer to customise the output format; they are enabled using the MAP directive which has the format:

MAP [[filename] ,option, option, ...]

The filename gives the name of the file to which the map file is to be written. The options specify which parts of the map file are to be written and all consist of single letters:

Option	Meaning
F	Produce a mapping of input files in the output file
H	Show where the input hunks (sections) were placed
L	Map the library placements
S	Show all external symbols
X	Show a cross reference of external symbols

When generating cross-reference information it is often useful to be able to separate this from the map file information. This can be done using the XREF directive which allows a separate crossreference file to be specified. It has the form:

XREF filename

To control, the layout of the map file several directives are available which are used in the same way as the more normal output directives or options:

FANCY	Enables usage of printer control characters in the map file.
FWIDTH n	Width of file names (default 16).
HEIGHT n	Lines on a page in map file, 0 indicates no pagination (default 55).
HWIDTH n	Width of hunk names (default 8).

INDENT n	Columns to indent on a line. This is included in width (default 0).
PLAIN	Turns off the FANCY map file option.
PWIDTH n	Width of program unit names (default 8).
SWIDTH n	Width of symbol names (default 8).
WIDTH n	Sets the maximum line length for the map and cross reference listings. This is useful when sending the output to a device which has different line length requirements. If not specified one width defaults to 80.

Options

Blink provides a large number of keywords to give the programmer a wide number of options. Although some of these may seem superfluous, the intention is to provide the programmer with as many options as possible, even if some of these options appear rather obscure.

BATCH	This causes Blink to supply the default value of 0 for all undefined symbols. Normally, Blink will pause after each undefined symbol to give you an opportunity to correct the error. If you specify the BATCH option, it will not pause.
BUFSIZE n	Sets the I/O buffer size for Blink. By default, all I/O is done in blocks as large as the available memory permits; this leads to extremely fast link times. This option may be useful if so little memory is available that the normal allocation scheme fails.
DEFINE symbol=val	DEFINE symbol=symbol This defines a symbol to be used in the linking process. This is particularly useful in conjunction with the PRELINK option to force certain routines to be pulled from the library even though no references to them exist. Note that you can assign either a value or another symbol.
IGNORE	Force Blink to continue after serious errors. Note that the use of this option may result in a nonexecutable file if an error has occurred.
NOALVS	Forces Blink to warn you when it creates ALVs to resolve 16 bit PC relative code. This can be Used to watch for Blink creating a non-relocatable object from what was intended to be relocatable code.

QUIET	Causes Blink to print out no messages unless an error occurs.
WITH file	Specifies a file containing Blink command options to be processed for this link. More than one WITH file may be specified and WITH files may contain WITH statements. The contents of all WITH files will be treated as if they were specified on the Blink command line.
VERIFY file	Specifies a file to contain all the messages output by the linker. If this is not specified all messages are written to the standard output stream. Note that VERIFY is a synonym for VERIFY.
VERBOSE	Causes Blink to print out the names of each file as it processes it and a summary of memory usage and elapsed time on completion.

'WITH' files

A WITH file provides a method for encapsulating long and complex (or short and simple) Blink command lines in a control file, known as a WITH file, traditionally with the extension LNK. The format of a WITH file is identical to the normal command line driven structure, except than line breaks may be used in place of spaces. For example the first simple command line example could have the WITH file:

```
FROM first .o+second.o
```

Consider a slightly more complex example of program which is to consist of two modules (object files) and a library:

```
FROM a.o+mine.o+hers.o      ; the object files
LIB amiga.lib                ; add a simple library
TO prog1                     ; output file name
ADDSYM                       ; add exported symbols
VERBOSE                       ; output messages during linking
MAP mine.map,F,H,X           ; produce map file
```

Note the use of the ; to delimit comments in a WITH file. This file can then be passed for execution to the linker using the command line (assuming the WITH file is saved as mywith.lnk)

```
BLINK WITH mywith.lnk
```

A more complex example would be to consider a mixed language example with both C and assembler modules. Assuming that the main project was written in C, the normal C runtimes would have to be included, additionally it may be necessary to force some external data items defined in the assembly language to use the `_` prefix required by C, this can be done using the `DEFINE` directive:

```
FROM LIB:c.o           ; C startup code
a.o+b.o               ; assembler object files
d.o+e.o               ; C object files
LIB LIB:lcm.lib        ; Math
LIB LIB:lc.lib         ; and C runtime libraries
TO prog2              ; output file
DEFINE _menu=menu      ; alias menu defined in assembly to menu
                      ; referenced in C
MAP a.map,f,h,l,s      ; map file
XREF a.xrf             ; separate cross-reference file
HEIGHT 66              ; longer page length
FWIDTH 10              ; narrower filename width
```

Note that the contents of `WITH` files are always processed after any files explicitly named on the command line, hence if the last `WITH` file were named `mywith2.Ink`, then the command line:

```
BLINK WITH mywith2.Ink LIB mylib.lib
```

would search the `mylib.lib` file before searching the `lcm.lib` or `lc.lib` files.

BLINKWITH: the Blink assign

The assignment `BLINKWITH`, if assigned, is taken by Blink to be the name of a `WITH` file whose contents is to be searched before any of the other files mentioned on the command line. This allows a template `WITH` file to be generated with the standard startup and library files mentioned in the `BLINKWITH` file, whilst the additional files are specified on the command line. The format of the `BLINKWITH` variable should be:

```
ASSIGN BLINKWITH: s:default.Ink
```

Special HUNK names

Whilst linking, Blink considers certain section names to be special. These are as follows:

NTRYHUNK

There can only be one of these and the hunk with this name will be the *first* hunk in the output executable.

__MERGED

Only data and BSS Hunks with this name are permitted. If a data or BSS hunk has this name it is merged with any other hunks with the same name and the __LinkerDB symbol set up to point to the area.

NOMERGE

A hunk with this name is *never* merged with any other hunk and will occupy a single, separate hunk in the final output file.

Reserved symbols

To provide access to the base of the sections created by the linker various symbols are invented by the linker. These are as follows:

RESBASE, _RESLEN, _NEWDATA1,

__BSSBAS, __BSSLEN

Reserved symbols. These are reserved symbols used in the SAS/C resident startup code. If you use them in your own code your programs are almost certain not to work correctly.

_LinkerDB

Pointer to static merged data section. The address of this external variable points to the base of the __MERGED data section.

Blink Messages

Whilst running Blink may discover things which it needs to bring to your attention. These may either be error messages or observations on the program which is being built.

Blink Warnings/messages

The messages in this section although warnings, will often indicate that the final program will be unusable in the form intended and you should not run it unless you are certain that you understand what you are doing.

Warning MERGED data > 64K

The merged data section has exceeded the limit of 64K. The problem may be rectified by moving some of your data in the `__MERGED` section into a far data section..

Warning! Absolute reference to <name>
module: <mod> file: <file>

An absolute reference was detected to a merged data item, whilst building a resident load module. This warning will only be given if a reference has been made to the symbol `__RESBASE`.

Warning: ALVs were generated

This message is generated when the `NOALVS` option is used, indicating that ALVs were generated.

Enter a `DEFINE` value for <name> (default stub) :
Undefined symbols... First Referenced

These messages indicate that the linker has encountered a reference to a symbol for which it cannot locate a definition. The second message is issued if the `BATCH` keyword is specified, whereas the first allows you to specify an alternate name for the reference.

Blink Errors

These are the errors which may be issued by the linker. In general errors may be ignored by use of the IGNORE keyword to Blink, however programs so produced may not function correctly. The error numbers are broadly divided so that 200-400 may be issued by either pass. 401-500 are issued by pass 1, whilst 501-599 are issued during pass 2.

Note that if a module has been compiled with standard line number debugging turned on then the line number on (or near) to where the problem occurred will also be reported.

200 Out of memory!

The linker does not have enough memory left to successfully complete the link.

300 System error <val> on read

A system error occurred whilst attempting to read from the disk. This should only occur if the disk has been damaged in some way. The value of the error is given by <val> .

301 System error <val> on write

A system error occurred whilst attempting to write to the disk. This will normally indicate that the disk is full. The value of the error is given by <val> .

400 * Break: Blink terminating**

This message is printed when the operation of the linker is interrupted by the user pressing Ctrl-C.

425 Cannot find library <file>

The file named in a LIB statement could not be located by the linker. This is probably due to a full pathname not being given for the file.

426 Cannot find object <file>

The file named in a FROM or ROOT statement could not be located by the linker. This is probably due to a full pathname not being given for the file.

443 '<file>' is an invalid-file name

The filename specified in a FROM, LIB or ROOT statement is invalid. Typically this will be because the name is null.

444 hunk_symbol has bad <val> symbol <file>

A hunk_symbol hunk type was encountered by the linker which did not have the external type set to zero, but instead to val. If this error occurs it indicates that the named input file was damaged in some manner.

445 Invalid hunk-symbol <name>

A hunk_symbol hunk type was encountered by the linker during parsing of the external definitions. The named symbol was attached to this hunk.

446 Invalid symbol type <val> for <file>

Whilst parsing external declarations an unknown symbol type <val> was encountered in the named file.

448 <file> is not a valid object file

The named file did not match the specifications for an object module.

449 No hunk end seen for <file>

On reaching the end of a hunk within the named file an end marker did not appear.

450 Object file <file> is an extended library

An attempt has been made to use a library as the operand of a FROM or ROOT statement. Libraries may only be searched, not included.

501 Invalid Reloc 8 or 16 reference

An attempt has been made to generate a branch between two differently named sections. Branches may only occur within a common section. This error will normally indicate an attempt to execute the data section!

502 <name> symbol - Distance for Reloc16 > 32768

The target of a 16 bit branch is more than 32K away from the reference. In general you should not see this message due to ALV generation.

503 <name> symbol - Distance for Reloc8 > 128

The target of an 8 bit branch is more than 128 bytes away from the reference.

504 <name> symbol - Distance for Data Reloc16 > 32768

A 16 bit base-relative data section access is attempting to reach more than 32K. This error will normally indicate you are very close to the 64K limit on near data, and a module has had its data section fall off the end of the merged data section (biased by 32K). The solution is to reorder your modules putting the ones with large data sections last alternatively you may have to move some of your data from the __MERGED section to a normal data section.

505 <name> symbol - Distance for Data Reloc8 > 128

An 8 bit base-relative data section access is attempting to reach more than 128 bytes. This error will normally indicate incorrect code generation from the compiler.

506 Can't locate resolved symbol <name>

During the second pass the linker could not locate the named symbol in its table. This will either indicate an internal linker failure or a damaged library file.

507 Unknown Symbol type ,<val> , for symbol <name>

During the second pass the linker could not match the type of the named symbol in its table. This will indicate an internal linker failure.

508 Symbol type <val> unimplemented

Whilst parsing external declarations an unknown symbol type <val> was encountered in the named file. Note that the equivalent error (446) is reported during pass 1.

509 Unknown hunk type <val> in Pass2

The named file did not match the specifications for an object module. Note that this message is identical to the pass 1 error 448.

510 <name> symbol - Reference to unmerged data item

A module has attempted to access an data item which the linker has not placed in the MERGED section using merged access type.

515 An ALV was generated pointing to data <name> symbol

An ALV was generated in the data section of the program. This will only occur if code generation has been performed in a data section, and as such this error will normally indicate an internal

compiler failure.

600 Invalid command '<cmd>'

The named command was not recognised by the linker.

601 <cmd> option specified more than once

An attempt has been made to specify a command, which may only appear once, more than once, e.g. attempting to specify two TO files.

602 Unable to open output file ' <file>'

The named output file could not be opened. This may be because the disk or directory is full.

603 <val> is not a valid number

The value <val> which appeared as a numeric argument could not be parsed as such.

604 with file is not. readable

An error occurred whilst reading the WITH file.

605 Cannot open with file '<file>'

The named WITH file could not be opened.

607 No FROM files specified

No FROM or ROOT files were specified so the linker cannot start linking.

608 Premature EOF encountered

End-of-file occurred unexpectedly. This will normally indicate serious file system structure problems.

609 Error seeking in file. <file>

An error occurred whilst attempting to seek about the named file. This will normally indicate serious file system structure problems.

611 Reloc found with odd address for symbol <name>, file <file>

A 16 or 32 bit relocation was attempted on a non word-aligned boundary. This is always illegal on the 68000.

ERROR: Invalid decimal constant '<val>'.

The value <val> which was entered in response to an undefined symbol was an invalid decimal constant.

ERROR: Invalid hex constant '<val>'.

The value <val> which was entered in response to an undefined symbol was an invalid hexadecimal constant.

ERROR: Multiply defined symbol '<name>'

A symbol has been redefined. The file in which it first appears is named, as is the file in which the attempted re-definition occurs.

ERROR: Symbol '<name>' is not defined.

The named symbol which was entered in response to an undefined symbol was also undefined.

Hunk #n not written

The numbered hunk n was not written to disk. This will indicate an internal linker failure.

Unknown internal error

An internal error occurred whose error number was not recognised. This indicates a serious internal linker failure.

Chapter 6

Other Tools

S-record splitter

If you are developing code for an embedded system, you will need to 'burn' your final code into EPROM. Most EPROMs are byte-wide devices, so if you are producing a system with a 16 bit data bus, you need two EPROMs, one for the odd numbered bytes and the other for the even numbered bytes. If you are developing for a system with a 32-bit data bus, four different EPROMs are required.

Many EPROM programmer's accept Motorola S-records as input and GenAm will happily generate absolute S-record code, but GenArn generates the entire output file in one go. This is fine for downloading into RAM or where the ROM size matches the data bus width, but doesn't help in the situation above. This is where SRSplit, the S-record splitter comes in. It splits an S-record file into two or more S-record files. The address fields of the new files are calculated so that they are appropriate for 'burning' into EPROM as described above.

SRSplit is a CLI program whose command line should be of the form:

```
<-options> <filename> [filename]
```

The filename gives the file to split; more than one split at once. The options available are

- p** this must be followed by the number of pieces into which the file will be split. The default is 2 and the most common alternative is 4, although other values (up to 9) may be used if you wish.
- b** this specifies a base offset that will be added before the address calculation is performed. The offset should be specified in decimal or preceded by 0x for hexadecimal. You can also use octal by specifying 0 at the start of the number. See below for an example.

Note that the options *must* come before the filename. The output filenames are the input filename with 1, 2 etc added.

Command line examples

`bootrom.mx`

This splits `bootrom.mx` into `bootrom.mx1` and `bootrom.mx2`

`-p4 test.mx`

This splits the file `test.mx` into 4 files: `test.mx1`, `test.mx2`, `test.mx3` and `test.mx4`.

`-b0x14000 -p4 test.mx`

This splits `test.mx` into 4 files, adding an offset of \$14000 to the addresses in the S-records. You might use this if your file is designed to be run at 0 but needs to be sent directly to an EPROM miner whose memory buffer starts at \$14000.

Operating system utility

Should you obtain new include files from Commodore you should be able to assemble them unchanged with GenAm – and you should also be able to pre-assemble these for fast assembly times.

However should you wish to use `_LVO` offsets to access any new library calls you will need to run the `FD2LVO` file below. Similarly if you obtain an FD file for a third party supplied library you can use this to generate the `_LVO` offsets.

`FD2LVO` performs the same task as `ConvertFD` program that was supplied with Devpac Amiga 2; the name has been changed to avoid conflict with the Commodore program that converts FD files to BASIC `.bmap` files.

FD stands for *Function Description*; LVO stands for *Library Vector Offset*.

FD2LVO details

FD2LVO converts Commodore FD files into library include files with extension `_lib.i` containing the `_LVO` offsets for inclusion into your programs.

The command-line is of the form

`FD2LVO [source] [destination]`

The source and destination file names should have their extensions omitted. The `-d` flag indicates that a whole directory is to be converted. For example:

`FD2LVO fdfiles/dos new/dos`

converts the single file `fdfiles/dos_lib.fd` to `new/dos_lib.i`

Appendix A

AmigaDOS Error Codes

This Appendix details the numeric AmigaDOS errors and their meanings.

103 insufficient free store

out of memory.

105 task table full

limit of 20 CLIs.

114 bad template

115 bad number

116 required argument missing

117 keyword requires argument

118 too many arguments

120 argument line invalid or too long

when using CLI commands.

121 file is not an object module

trying to execute non-executable file.

122 invalid resident library during load

201 no default directory

202 object in use

such as a file by another program.

203-object already exists

204 directory not found

205 object not found

most commonly a file.

206 invalid window

in name specification.

207 object too large

208 invalid action

209 packet request type unknown

210 invalid stream component name

name too long or contains control chars.

211 invalid object lock

212 object not of required type

such as directory name instead of file

213 disk not validated

disk is still being validated, or bad.

214 disk write-protected

215 rename across devices attempted

216 directory not empty

when trying to delete it.

217 too many levels 218 device not mounted

after specifying a volume name.

219 seek error

219 seek error

220 comment too big

file comments must be less than 80.

221 disk full

222 file is protected from deletion

223 file is protected from writing

224 file is protected from reading

225 not a DOS disk

226 no disk in drive

232 no more entries in directory

233 object is soft linked

234 object is linked

235 bad hunk

236 not implemented

240 record not locked

241 lock collision

242 lock timeout

243 unlock failed

303 Buffer overf low

in internal or user buffer.

304 break received

305 file not executable

E bit is cleared.

Appendix B

GenAm Error Messages

GenAm can produce a large number of error messages, most of which are pretty well self explanatory. This appendix lists them all in alphabetic order, with clarifications for those which require them.

Please note that GenAm is continually being improved and this list may not agree exactly with the version you have, there may be additional messages not documented here.

Errors

If you get a message beginning with INTERNAL please tell us - you should never see these.

probably missing

You have used an absolute reference where an immediate one was more likely. This will also occur if you miss out the address register when you are accessing variables via a base register. If you really mean the absolute reference, use an explicit .L or .W or disable OPT CHKIMM.

.W or .L expected as index size

absolute expression MUST evaluate

absolute not allowed

additional symbol on pass 2

somehow a symbol has appeared during pass 2 that did not appear during pass 1.

address register expected

addressing mode not allowed

addressing mode not recognised

assembly interrupted

bad floating point expression

BSS or OFFSET section cannot contain data

OFFSET sections and BSS sections can only contain DS directives.

cannot create binary file

could be a bad filename, or a write-protected disk, etc.

cannot export symbol

cannot import symbol

cannot nest MACRO definitions or define in REPTs

macro definitions may not be nested or defined within repeat loops.

cannot nest repeat loops

colon (:) expected

in multi-register 68020 argument.

comma expected

data register expected

data too large

DCB or DS count must not be negative

division by zero

duplicate MODULE name

module names must be unique.

error during listing output

listing will be stopped at this point.

error during writing binary file

normally disk full.

error in command line symbol

executable code only

only executable code may be assembled to memory.

expression mismatch

normally a syntax error within an expression.

fatally bad conditional

there were more ENDCs in a macro than I Fs.

file not found

floating point constant not allowed

floating point constant too large

floating-point register expected

after floating point instruction.

forward reference

garbage following instruction

hex floating point number too large

illegal BSR.S

a BSR.S to the following, instruction is not allowed change it to BSR.

Illegal type combination

immediate data expected

imported label not allowed

includee file read error

instruction not recognised

invalid 68020 addressing mode

invalid bitfield specification

invalid floating point expression

invalid FORMAT parameter

invalid IF expression, ignored

invalid index scale

invalid k-factor

invalid MMU function code

invalid MOVEP addressing mode

invalid number

invalid numeric expansion

the symbol is not defined or relative or a syntax error.

invalid opcode size for data/address register

invalid option

invalid printer parameter

invalid radix

invalid register list

invalid section name, TEXT assumed

invalid section specified

invalid section type

MMU register expected

privileged instruction

you have used an instruction that can only be used in supervisor mode after an OPT USER directive.

invalid pre-tokenised file

Either the pre-tokenised file itself is corrupt or it was produced with an earlier or later version of the assembler. Re-make the file from the include file using the Output Symbols command.

invalid size

line malformed

linker format restriction

the DRI format is restrictive about where it allows imports.

local not allowed

maths co-processor required

missing close bracket

missing ENDC

there were more IFs than ENDCs.

Missing quote

Misuse of label

MMU register expected

not yet implemented

number too large

odd address

not yet implemented

number too large

odd address

only (An) allowed for this instruction

only occurs with 68040 MMU instructions

only FPIAR allowed

option must be at start

ORG/RORG not allowed

out of memory

phasing error

should never happen, unless you have symbols that evaluate to different values on different passes. Investigate immediately before the first such error.

privileged instruction

when OPT USER is in operation.

program buffer full

change the program buffer size when assembling to memory.

register expected

relative not allowed

relocation not allowed

repeated include file

each include file may only be included once on each pass.

short branch cannot be 0 bytes

source expired prematurely

within an IF, MACRO or REPT and the source ran out.

spurious ENDC

spurious ENDM or MEXIT

spurious ENDR

symbol defined twice

symbol expected

undefined symbol

user error

caused by a FAIL directive

wrong processor

XREFs not allowed within brackets

Warnings

.L converted to .W

when optimising.

68010 instruction, converted to MOVE SR

MOVE CCR, is *not* a 68000 instruction (only when in 68000/8 mode).

ADD/SUB converted to LEA

when optimising.

base displacement shortened

when optimising.

branch could be short

forward branch could be optimised

branch made short

by optimising

directive ignored

invalid LINK displacement

if negative or odd.

LEA converted to ADDQ/SUBQ

misuse of register list

A register list created using EOUR has been used in an expression.

MOVEQ substituted

Move.l #nn, d0 optimised to moveq #nn, d0.

no ORG specified

When generating absolute code via S-records.

offset removed

xx(An) form reduced to (An) by optimising.

outer displacement shortened

when optimising.

quick form used

when optimising.

relative cannot be relocated

short branch converted to NOP

when optimising.

short word addressing used

when optimising.

sign extended operand

data in MOVEQ needed sign extension to fit.

size should be .W

Appendix C

Calling the Operating System

Introduction

The Amiga@ operating system is arguably with the exception of OS/2 the most sophisticated on any mass-produced computer, and is also the most complicated. The whole machine is based around the concept of libraries, which are essentially groups of subroutines (or functions to C-programmers) indexed off a large jump block. This Appendix is intended to explain the basics of calling libraries from assembly-language, and to give an idea of what each can be used for. One small Appendix ' cannot possibly describe the whole operating system, it is only meant as an introduction. For further information turn to the Bibliography for recommended books.

Please note that this was written with version 1.3 of Kickstart in mind. Full documentation for Release 2.0 of the operating system can be obtained directly from HiSoft and complete include files are already supplied on your Devpac disks.

Libraries

The most basic library is the exec library, which has to be called to open any other library, among other things. As with all libraries, a *library base pointer* is required to access it, and this must be loaded into register A6 before calling any function. The exec library is unique in that it doesn't have to be opened to obtain a base pointer - this can be obtained from the longword at location 4 - the only location in the whole machine guaranteed to remain the same in the future.

The base pointer so obtained can then be used to open further libraries, to obtain other library base pointers, and so on. Note that most libraries require A6 to be contain the base pointer for correct, operation (as they call other routines in the same library) though not all do. Parameters are passed to library routines in registers, and as a general rule registers D0/DI/A0/AI should be assumed to be corrupted by any call.

A large number of Include files are supplied with Devpac Amiga to allow easy access to the various parts of the operating system. These include files contain macro definitions, library offset symbols, data structure definitions, and bit field symbols. There now follows a library table showing the names of many of the components, and where the definitions can be found in the include directory.

File: this shows the file that contains the macro definitions and _LVO (Library Vector Offsets) symbols for calling the library.

Name macro: this is normally consists of a DC.B statement defining the ASCII for the name, ending in a null.

Base pointer: this is the symbolic name of the longword used for storing the base pointer. It always starts with an underline character, though when used from most C compilers this underline is not shown.

Calling macro: this is the name of the macro that calls a particular library. Note that this will corrupt register A6 as it will be loaded with the relevant library base pointer.

The following list gives the library followed by where its file can be found, the name macro,' base pointer and calling macro, in order:

- diskfont, libraries/diskfont_lib.i,
DISKFONTNAME, _DiskfontBase, CALLDISKFONT
- dos, libraries/dos_lib.i, DOSNAME, _DOSBase,
CALLDOS
- exec, exec /exec_lib.i, EXECNAME, _SysBase, CALLEXEC
- expansion, libraries/expansion_lib.i,
EXPANSIONNAME , _ExpansionBase, CALLEXP
- graphics, graphics/graphics_lib.i, GRAFNAME,
_GfxBase, CALLGRAF

- `icon`, `workbench/icon_lib.i`, `ICONNAME`, `_IconBase`, `CALLICON`
- `intuition`, `intuition/intuition lib.i`, `INTNAME`, `IntuitionBase`, `CALLINT`
- `mathffp`, `math/mathffp_lib.i`, `FFPNAME` -`MathBase` , `CALLFFP`
- `mathdouble`, `math /math ieedoubbas lib.i`, `IEEEDOUBNAME`, `MathIeeeDoubBasBase`, `CALLIEEEDOUB`
- `mathtrans`, `math/mathtrans_lib.i`, `MATHTRANSNAME`, `MathTransBase`, `CALLMATHTRANS`
- `translator`, `libraries/translator_lib.i`, `TRANSNAME`, `TranslatorBase`, `CALLTRANS`

For example, to call the `exec` library function `OpenLibrary` suitable assembler source code would be

```
CALLEXEC OpenLibrary
```

This macro is expanded into

```
move.l (_SysBase).w, a6
jsr    _LV0OpenLibrary(a6)
```

Diskfont Library

This is a library for handling fonts that are normally resident on the disk.

Files: `libraries/diskfont.i` and `diskfont lib.i`

DOS Library

One of the most straightforward of the libraries to use, this handles file I/O (Input/Output) to devices, including disk and console. It has some slight anomalies, notably addresses have to be passed in data registers, and many pointers have to be BCPL-type (i.e. longword aligned and divided by 4).

Files: `libraries/dos.i`, `dos_lib.i` and `doesextens.i`. The Release 2.0 includes have a separate `DOS` directory containing various new files although the above two are still provided for compatibility.

Exec Library

This is, the lowest level of library, responsible for things like memory management, library calls and message passing. The library never has to be opened - its base pointer is contained in location 4 although it is usual for programs to copy this into their own data area for maximum speed.

Files: `exec/ables.i`, `alerts.i`, `devices.i`, `errors.i`, `exec.i`, `execbase.i`, `execname.i`, `exec_lib.i`, `funcdef.i`, `initializers.i`, `interrupts.i`, `io.i`, `libraries.i`, `lists.i`, `memory.1`, `nodes.i`, `ports.i`, `resident.i`, `strings.i`, `tasks.i`, and `types.i`

Graphics Library

This is responsible for controlling exactly what appears on your monitor, including things like drawing lines, printing text, controlling RastPorts, sprite handling and fonts.

Files: `graphics/clip.i`, `copper.i`, `display.i`, `gels.i`, `gfx.i`, `gfxbase.i`, `graphics_lib.i`, `layers.i`, `rastport.i`, `regions.i`, `sprite.i`, `text.i`, `view.i`

Icon Library

This is responsible for handling the icons displayed by the Workbench.

Files: `workbench/icon.i`, `icon_lib.i`

Intuition Library

This library is the largest and is responsible for the windowing Intuition user interface. It has a very large number of functions, including those for window control, screens, gadgets, requesters, and event handling.

The main file is large and also includes a large number of other files, so don't be surprised if it takes a little while to read it all. It can be worthwhile to create your own specialised version without the less-often used constants, which can reduce the amount of other include files required.

Files: intuition/intultlon.i, intuition_lib.i

Maths Libraries

There are three maths libraries, all based on the official Motorola routines. The FFP (Fast Floating Point) library uses an 8-bit exponent, 24-bit mantissa format. The format used was designed for the 68000 series, and is exclusive to Motorola. The IEEE double library offers double-precision using the IEEE standard formats for numbers, and the Transcendental library is used for FFP trig and other functions.

Files: math/ mathffp_lib.i, mathieeedoubbas_lib.i,
mathtrans_lib.i

As a rule you should use GenAm in case sensitive mode (the default) when using the supplied Include files. Note that every Include file always includes any others it needs automatically, so you don't need to worry about it.

Release 2.0 libraries

A number of new libraries were added with Release 2.0 of the operating system.

GadTools provides simpler creation and control over the standard gadget types similar to those used by the Devpac editor.

Utility contains a number of convenient functions including those controlling the use of *tag lists* (simply pairs identifier numbers and corresponding values held as longwords in memory, terminated by a zero).. Such lists may be passed to new DOS, Intuition and Graphics library functions and extend the functionality of many fixed 1.3 data structures.

IFFParse, a disk loaded library also available for 1.3 which contains extremely flexible and powerful routines for handling the reading and writing of any IFF files. IFF is the Amiga Interchnage File Format used for pictures, sound samples, animations, formatted text and much more. It is also the format recognised by the Amiga® clipboard. For further information, refer to the IFF Manual mentioned in the *Bibliography*.

Commodities; using this library is the preferred way of augmenting the operating system by the addition of utilities such as 'hot key' programs, screen blankers, automatic window activation etc.

Workbench is now a callable library giving access to new features such as the application Tools menu, windows which are notified of icons being dropped into them and application controlled icons (these are known as AppMenus, AppWindows and Applcons).

ASL requester library is the standard Amiga® requester library providing file and font requesters. The Devpac editor is an example of a program which uses the ASL library.

Many of the Release 2.0 enhancements have gone into already existing libraries, with DOS and Intuition containing the main additions. It is now possible to call the DOS library for pattern matching, packet communication with handlers, standard command line argument processing, assignments, manipulation of path and file name strings, file record locking and notification. Note that some of these facilities, along with a file requester, are available under 1.3 via a third-party library called the ARP library.

The main new features of the Intuition library are control of public screens (Amiga® screens which can be shared amongst any number of applications; the Workbench is an example of a public screen), BOOPSI object-orientated gadgets. These provide a new and extensible way of handling the regular Intuition gadgets, images etc. with the additional ability of allowing creation of your own custom gadget types which are handled by Intuition.

Full and comprehensive documentation for these and all other Amiga® subsystems can be found in the Third Edition of the Amiga ROM Kernel Manuals and The AmigaDOS Manual available directly from HiSoft.

Example Programs

To help you get started programming AmigaDOS from assembly language we have provided the source to a few example programs in the Examples drawer.

demo.s

This is the program used for the tutorial at the beginning of the manual. It uses the DOS library to print a message on the current CLI window.

freemem.s

This is a program that uses Intuition to create a window in which the system free memory is constantly displayed, until the close gadget is clicked on.

helloworld.s

This is the assembly-language conversion of the C program *Final Version of the Simple Program* described at the beginning of *Intuition - The Amiga User Interface* (revision 1.0 page 2-9). The conversion has not been optimised in any way, for example the structure assignment for `NewScreen` would be more efficient using `dc.w/dc.l` statements, but has been left as `MOVE` instructions for a more accurate conversion. The program opens up a custom screen and, within it, a window with a simple message.

CLI vs Workbench

There are two program environments on the Amiga@ - the windows & icon driven Workbench, and the CLI or Shell. Devpac Amiga runs under either interface although most of the example programs can only run from a command line environment, the difference being in the startup and exit code.

CLI Startu

When a program is run from the CLI it starts with register A0 containing the address of the command line, and D0 containing its length. The DOS handles returned by Input and output can be used for I/O with the console device and to exit, the program should simply RTS.

Workbench Startup

When a program is run from the workbench it has to wait firstly message, and on terminating it has to reply to the message (after doing a Forbid call) before RTSing. The DOS functions Input and output will not return valid handles, so you will need to open a window to perform any console I/

The startup differences are detailed in Chapter 30 of the *Amiga ROM Kernel Manual: Libraries and Devices*, together with the assembly language source of the startup code used by C programs.

A skeleton version of this for assembly language programmers can be found in the file `misc/easystart.i` which is included by the `freemem2.s` example program. It should be included at the very front of your .programs, and handles the message-passing to allow programs to be run from the Workbench. Of course to do this you will need to create an icon for your program, using Create Icons? on the editor Settings menu.

Other 68000 Series Processors

When writing commercial programs for the Amiga® it should be borne in mind that some users can have 68020, 68030 or even 68040 processors in their machines instead of the regular 68000. One problem is the `MOVE SR` instruction which has become privileged in the 010 processor and upwards. To get around this, the `exec` library has a function called `GetCC` which returns the condition codes in register `D0`. It uses no other registers and doesn't actually need the library base pointer in `A6`. For example, if you want to save the condition codes on the stack, instead of doing

```
move    sr, -(sp)
```

you should do

```
movea.l 4.w, a0      get base pointer
jsr      _LVOGetCC(a0) call GetCC
move.w   d0, -(sp)    put on stack
move     d0, ccr       and set codes
```

This uses `A0` as a scratch register. Its best not to use a macro for this call in case you inadvertently change the condition codes (`movea` leaves them alone).

Another difference is that exception stack frames are very much bigger with 68010, 020 processors etc., but this should only affect debuggers and the like.

Appendix D

Using the CLI

Introduction

Like most other programming development tools, Devpac Amiga can be run from the Amiga® CLI (or Command Line Interface). Although you may choose to use the simpler icon driver Workbench interface, this appendix gives some advice which will help you to understand the operation of the CLI.

Files, Volumes and Directories

AmigaDOS files can be stored on floppy disks, the RAM-disk or a hard disk. As floppies can hold over 800K of data and hard disks even more, AmigaDOS supports *directories*, which are named sections of disks that can contain files and other directories (when using the Workbench, directories are called *drawers*). To specify a file within a directory the / symbol is used, so if you want to edit a file called `test.s` which is in a directory called `source` the command is:

```
devpac source/test.s Return
```

Note that the CLI doesn't care about the case of the letters in commands or filenames - if you had typed:

```
devpac SourCe/Test.S Return
```

it would do just the same. The way Devpac, GenAm and MonAm are invoked is similar to most other CLI commands - they consist of a *command* section (Devpac in this case) and, optionally, something following it, known as a *command line*. If you want to know what files you have on the disk, enter the command

```
dir      Return
```

which will show them all (`dir` stands for *show directory*).

AmigaDOS can take a little while to do, so be patient. If there are any directories on the disk they will be shown first, followed by (dir), then all the filenames in alphabetical order.

Incidentally, as AmigaDOS doesn't care about the case of filenames, you may wonder how it knows the format in which to display the directory. The answer is that the exact name specified when the file (or directory) was first created

The CLI (or Shell) keeps track of the *current* disk and directory; the cd command can be used to check the current state and to change it. After initially booting up with a backup of the Devpac Amiga disk the current disk will be DF0: (DF0: is the internal drive, DF1: is the external one), and you can see this by typing

```
cd      Return
```

Typing cd on its own like this will always show you the current directory. If you want to change to another directory, for example the one containing the examples programs you follow the cd command with the directory name, e.g.

```
cd examples  Return
```

If you now type dir you will see a list of files, or typing cd on its own will show DF0: examples To look at text files, which is what the files ending in .s are, you can enter what the files endin

```
type demo.s  Return
```

which will show the contents of demo.s on the screen. If you want to pause this you can hold down the right mouse button, or you can stop it altogether by pressing Ctrl-C.

To change back to the previous directory you could use

```
cd df0:Return
```

or, alternatively

```
cd : Return
```

The : on its own means the top level directory on the current disk, or root. If we now want to look at demo.s once more we could change directories again, but an easier way would be to specify the directory together with the filename, e.g.

```
type examples/demo.s Return
```

Note the way the / is used.

AmigaDOS Wildcards

AmigaDOS has a somewhat different idea about wildcards in filenames from other operating systems. They are very powerful and can do wonderful things, but obvious-to-use they are not. Here is a summary of them:

?	any single character
%	the null string
# <P>	one or more occurrences of the pattern <p>
<p1 > <p2>	either pattern p1 or pattern p2
()	groups patterns together

The common uses for these are ? which is straightforward enough and #? which means all files and is analogous to * in MS-DOS and CP/M.

Device Names

All I/O devices on the Amiga have names by which you can refer to them in CLI commands and from within programs. These all have short names ending in a and in fact you have already met two of them - DF0: and DF1: ; the two floppy drives. Here are some more.

RAM:

This device is a RAM-disk which can be used much like a regular floppy but is much faster to save and load as it stores everything in memory.

The disadvantage is that all data will be lost when you switch off or reset (or if one of your programs goes a little crazy). It is dynamic in size, allocating memory from the system pool when it needs it.

RAD:

RAD: is a RAM-disk that differs from RAM: in two important ways:

- Its size is fixed at mount-time. See your system documentation for further details.
- It will survive a warm boot with Ctrl-Left Amiga-Right Amiga and you can re-boot from it using Kickstart 1.3 or above.

PAR:

This is the parallel port, usually only for output.

SER:

This is the serial port, which can be used for both input and output, normally to printers and modems.

PRT:

This is the general printer device, which is configured by the user with the Preferences program. Accessing the printer via this device means that programs don't need to know which port is being used and what sort of printer is connected.

NIL:

This is a dummy device, which throws away all input to it. It will generate end-of-file when any attempt is made to read from it.

CON:

This is the console device, and creates a window for keyboard I/O. The window specification should follow the CON: in the name and be of the form

CON:x/y/width/height/title

RAW:

This is, to quote the AmigaDOS User's Manual, "intended for the advanced user". Do not use RAW: experimentally."

*

This is not actually a device, but can be used in the same place as filenames and means the *current CLI window*.

NEWCON:

The 1.3 version of the operating system also includes a NEWCON: device which gives all the functionality of CON: with the added features of a command history buffer and more powerful line editing. Release 2.0 now includes all of this in the standard CON: along with new keywords such as CLOSE, AUTO and SCREEN. For example,

```
CON:O/25/640/150/Diagnostics/CLOSE/AUTO/WAIT/SCREEN
DEVPAC.1
```

AmigaDOS Commands

To help you use the CLI there now follows a brief description of the most commonly-used commands, together with examples. It is not a complete description and if further details are required the *AmigaDOS User's Guide* is recommended for further reading.

[] s denote required parameters, < >s denote optional ones. Most commands support redirection of output using > at the beginning of the command line, so for example to dump a directory to a file you could use

```
dir >allf files.txt
```

ASSIGN <name> <directory>

This allows you to use special names for, particularly, directories and initially the system sets up the following assignments (where boot is used to denote the name of the boot disk):

Name	Default	Use
C:	boot:c	CLI looks here for all commands entered
L:	boot:l	system handlers
S:	boot:s	startup sequence
LIBS:	boot:libs	libraries not in ROM
DEVS:	boot:devs	device drivers
FONTS:	boot:fonts	fonts not in ROM
SYS:	boot:	boot disk

As well as changing these you can add your own, so if you wanted an easy way of getting to the graphic include files in the first external drive you could enter the command

```
assign graf: df1:include/graphics
```

then access your files (regardless of the current directory) with commands like:

```
type graf:rastport.i
```

To show all current assignments (and devices) don't specify any parameters in the command and to remove a definition don't specify a directory.

CD <directory>

This either changes the current directory to the supplied parameter, or if there is no parameter, displays the current directory. Workbench 2.0 users can omit the CD command and just name if desired.

COPY [oldname] <TO> [newname] <ALL> <QUIET>

This copies files individually, in groups or complete directories. The ALL option copies directories, and the QUIET option disables the filename list produced for each copy. Any existing files with the new name will be deleted first without asking. It is best illustrated by examples:

copy examples/demo.s to ram:demo.s

copies an individual file.

copy examples to df1:genexamples all

copies the complete contents of the directory into the other directory which will be created if it does not already exist.

copy include to df1:include all

copies the include directory, as well as all those within it, on to the external drive, creating the new directories if it needs to.

copy #?.info ram:

copies any files ending in .info onto the RAM-disk

DATE <date> <time>

This shows or sets the date and/or the time. The format for the date is DD-MMM-YY, and that for the time is HH:MM. Either or both the time and date can be set, but if no parameters are given the current date and time will be displayed. Example:

date 25-jan-87 12:00

DELETE [name] <name> <name> . <ALL>

This can delete individual files, a list of them, or whole directories. For safety, if you specify a directory name, it will only be deleted if it is empty - you must give the ALL option to delete its contents first, e.g.

delete demo.s demo

delete examples all

DIR <name>

This shows the files in the current directory (if none specified) or the directory specified.

DISKCOPY [drive] TO [drive] <NAME name>

This does a complete copy of one floppy disk to another. The destination does not need to be formatted and any files on it previously will be destroyed. If the same drive is specified in both parameters you will be prompted to swap disks when necessary. It is strongly recommended that you write-protect the source disk before running this program to ensure you cannot accidentally lose data. By default the copy has the ' same name as the original, but the NAME option can override this.

```
diskcopy df0: to df1:
```

```
diskcopy df0: to df0: name "Backup"
```

ECHO <string>

This command prints a string on the screen and is mainly useful in startup sequences or EXECUTE batch files. Example:

```
echo "Devpac Amiga Workbench disk"
```

ENDCLI

This terminates the current CLI. Be careful with this - if you execute it from the last CLI without Workbench loaded, it will end and you will not be able to do anything subsequently. It is intended for use in CLIs created with the NEWCLI command.

EXECUTE [name] <arguments>

If you have to do a very repetitive command sequence from the CLI it can be easier to create a text file containing the commands, then EXECUTE the text file and all the commands will be done for you. Parameters get substituted (like macro parameters in GenAm), so if the text file `dolt` contained this

```
.key file /a
```

```
copy <file> to ram:copy ram :<file> to prt:
```

```
delete ram:<file>
```

then the command

```
execute doit demo.s
```

would copy the file `demo.s` to the RAM-disk, copy it from there to the printer, then delete the RAM-disk copy when it had finished. The `.` key is used to specify parameters, `f` file in this case, the `/a` means it is always required.

FAULT [number]

This command displays the meaning of any given AmigaDOS error number.

FORMAT DRIVE [drivename] NAME [diskname]

This command initialises a floppy disk, *destroying everything that was on there previously*. The drive name should be `df0:` or `df1:` as required and the `diskname` is the name assigned to that disk. The full command must be typed to save you from accidentally erasing a disk. Example:

```
format drive df1: name "My backups" INFO
```

Shows the status of each drive and mounted disk, including the free space and its name. The free space is shown in blocks, which are normally 512 bytes, so half of the number gives how many Kbytes are free.

JOIN [fuel] l<file2 etc> AS [newfile]

This copies up to 15 files together into a new file e.g.

join part1 part2 as allparts

LIST <name>

This shows the length and creation date of each file within a directory.

LOADWB

This command brings up the icon-driven Workbench interface in the screen behind the CLI window. Workbench disks supplied with machines have this in the startup file so it is done automatically.

MAKEDIR [directory]

This command creates new directories. No file or directory of the given name should exist already.

NEWCLI <window>

This command creates additional CLIs which run as concurrent programs. You can specify the window size of the new CLI if you want to in the manner of a CON: device window.

RENAME [oldfile] <TO> [newfile]

This command changes the name of a given file on a disk. It can also be used to move a file between directories (though not between disks). Examples:

rename demo.s to demo.bak

rename include/misc/start.i to example/startup.s

RUN [command]

This allows commands or programs to run as concurrent tasks, i.e. run at the same time. It does this by creating another CLI (without a window unless redirection is specified) and passing the command on to it. When it finishes it removes the CLI. If you wish to RUN a sequence of commands you should follow the end of each line with a + sign e.g.

```
run copy demo.s to prt:
```

will copy a file to the printer in the background, while:

```
run copy demo.s to prt: +  
echo "File printed"
```

will do as above, but additionally echo a message after the copy command finishes.

STACK <size>

This command sets the size of stack allocated to each command or program run from the CLI, or displays it if no size is specified. The default value is 4000 bytes which is normally enough for all the described CLI commands and our programs, with the possible exception of `d i r` on heavily-nested directories. It is also possible that other programs (such as a C compiler) will require additional stack, and this should be documented in the manual for such programs.

TYPE [name] <OPT N> <OPT H>

This command is normally used to show a text file on the screen, the output of which can be paused by pressing the right mouse button. If you print a non-text file it is possible that you may change character sets and all typing will then appear to be gibberish. To correct this press:

```
[Return] [Ctrl]-0 [Return]
```

The N option will add line numbers to the display, and the H option will show the hex format of the file as well as the ASCII, similar to dump programs on other machines.

WHY

This command can be used after a CLI command has failed and will normally tell you the reason for the failure.

Startup Sequence

When you boot from a Workbench disk the file `s/startupsequence` is read and all commands contained within it executed. On Workbench disks supplied with Amigas and the Devpac Amiga disk this is used to load the Workbench program itself. If you want to edit this file you can use Devpac by entering

`devpac s:startup-sequence`

(using `s :` means it will find it no matter what the current directory is). You can change this to your requirements, then save it out. The new sequence will then be executed when you next boot.

Appendix E

The Floating Point

Co-processor

This Appendix is designed to give a quick overview of the 68881/68882 maths co-processor's registers and formats as the Motorola M68000 Family Programmer's Reference Manual lacks this, although it does include full details of the co-processor instructions.

The FPUs contain 8 data registers, named FP0 - FP7, each of which stores an 80 bit extended format number, and three control registers, the floating point control register (FPCR), floating point status register (FPSR) and floating point instruction address register (FPIAR).

Although the floating point data registers always store 80 bit extended precision numbers, the chip can convert these to and from a number of different formats as detailed below:

Extended precision

Extended precision format is stored in memory as 12 bytes. The bit layout is:

95	94-80	79-64	63-0
Sign	Exponent	zero	Mantissa

The sign bit is 0 for positive numbers and 1 for negative numbers. The mantissa has an implied binary point at bit 63 and thus ranges in value from 1.0 to $<2:0$.

The exponent is held in excess 16383 ($\$3FFF$) format with values of 0 and $\$7FFF$ being treated specially.

When the exponent is \$7FFF, the value represents Not-A-Number (NaN), the type of which is determined by the mantissa. Zero mantissas indicate infinity (∞), whilst non-zero mantissas indicate other NaN conditions.

With an exponent of 0 there are two possibilities. The number zero is represented by all bits zero, whereas other values are denormalised numbers with an exponent of -16383 (\$3FFF).

Double precision

The double precision IEEE format represents a number in 8 bytes. The bit layout is:

63	62-52	51-0
Sign	Exponent	Mantissa

The sign bit is 0 for positive numbers and 1 for negative numbers. The mantissa has an implied binary point at bit 51 and thus ranges in value from 1.0 to <2.0 .

The exponent is held in excess 1023 (\$3FF) format with values of 0 and \$7FF being treated specially.

When the exponent is \$7FF, the value represents Not-A-Number (NaN), the type of which is determined by the mantissa. Zero mantissas indicate infinity (∞), whilst non-zero mantissas indicate other NaN conditions.

With an exponent of 0 there are two possibilities. The number zero is represented by all bits zero, whereas other values are denormalised numbers with an exponent of -1022 (\$3FE).

Single Precision

The single precision IEEE format represents a number in 4 bytes. The bit layout is:

31	30-23	22-0
Sign	Exponent	Mantissa

The sign bit is 0 for positive numbers and 1 for negative numbers. The mantissa has an implied binary point at bit 23 and thus ranges in value from 1.0 to <2.0.

The exponent is held in excess 127 format with values of 0 and \$FF being treated specially.

When the exponent is \$FF, the value represents Not-A-Number (NaN), the type of which is determined by the mantissa. Zero mantissas indicate infinity (∞), whilst non-zero mantissas indicate other NaN conditions.

With an exponent of 0 there are two possibilities. The number zero is represented by all bits zero, whereas other values are denormalised numbers with an exponent of -126.

Packed Decimal

Packed decimal numbers are stored is stored in memory as 12 bytes. The bit layout is:

Bit	Meaning
95	Sign of mantissa.
94	Sign of exponent.
93-92	If %11 (i.e. both bits set) then a NAN or infinity (∞). Otherwise 0. See below.
91-80	3 least significant digits of exponent in decimal.
76-79	Most significant digit of exponent in decimal on a FMOV.P to memory if a fourth digit is required; otherwise don't care.
75-68	Don't care.
67-64	Most significant digit of the mantissa.
63-0	Remainder of digits of the mantissa.

If bits 93 and 92 are both one then bits 91-80 (the exponent) will be \$FFF. If bits 63 to 0 are all zero then this represents infinity (bit 95 giving the sign) otherwise the value is Not-A-Number (NaN).

We will now discuss the floating point control registers.

FPCR Floating point control register

Although this is a 32 bit register only the bottom two bytes are defined as yet. The more significant byte is known as the *FPCR Exception Enable Byte* and controls whether particular conditions will cause an exception (if the corresponding bit is one) or whether the appropriate bit in the FPSR exception status byte is set. See below. The bits are as follows:

Bit	Name	Meaning
8	INEX1	Inexact decimal input
9	1NEX2	Inexact operation
10	DZ	Divide by zero
11	UNFL	Underflow
12	OVFL	Overflow
13	OPERR	Operand error
14	SNAN	Signalling Not-A-Number (NaN)
15	BSUN	Branch/set on unordered

The least significant byte selects the rounding mode and rounding precision and is known as the *FPCR Mode Control Byte*. It is laid out as follows:

Bits	Name	Meaning
3-0		Zero
54	ROUND	Rounding direction. Towards: 00 nearest 01 zero 10 minus infinity 11 plus infinity
7-6	PREC	Rounding precision: 00 Extended 01 Single 10 Double 11 Reserved

FPSR Floating point status register

This is a 32 bit register, which is divided into four bytes:

31-25	24-17	16-8	7-0
condition code	quotient	exception status	accrued exception

The *FPSR Condition Code Byte* is updated after all the floating point instructions whose destination is a single data register F P 0 - 7 other than FMOVM. The bits are as follows:

Bit	Name	Meaning
24	NAN	Not a number
25	I	Infinity
26	Z	Zero
27	N	Negative
31-28		Always 0

The quotient byte contains the sign of the quotient (bit 24) and 7 least significant bits (bits 23-17) of the quotient after an FMOD or FREM instruction. This is normally used as the first stage of performing approximations to trigonometric functions by taking the remainder after a division by a fraction of pi.

The *FPSR Exception Status Byte* (EXC) is updated after all the floating point instructions whose destination is a single data register FP0-7 other than FMOVM. The bits are as follows:

Bit	Name	Meaning
8	INEX1	Inexact decimal input
9	INEX2	Inexact operation
10	DZ	Divide by zero
11	UNFL	Underflow
12	OVFL	Overflow
13	OPERR	Operand error
14	SNAN	Signalling Not-A-Number (NaN)
15	BSUN	Branch/set on unordered

In the *FPSR Accrued Exception Byte* (AEXC) the bits are 'sticky' i.e. only cleared by an explicit move into the FPSR. is updated after all the floating point instructions whose destination is a single data register FP0-7 other than FMOVEM. In the table below, the Exception status bits column gives the condition in the FPSR Exception Status byte that will cause the appropriate bit to be set:

Bit	Name	Exception status bits	Meaning
2-0			Always 0
3	INEX	INEX1!INEX2!OVFL	Inexact
4	DZ	DZ	Divide by zero
5	UNFL	UNFL&INEX2	Underflow
6	OVFL	OVFL	Overflow
7	IOP	BSUN!SNAN!OPERR	Invalid operation

In the table above, ! means OR and & meaning AND. Thus bit 3 of the AEXC will be set after an instruction if it was already set or if the INEX1, INEX2 or OVFL bits in the EXC byte get set.

FPIAR Floating point instruction address register

The floating point co-processor stores the current program counter in the Floating point instruction address register when it starts to process an instruction , so that exception handlers can determine the instruction that cause the exception. The handler cannot just look at its own program counter as most of the FP instructions are executed concurrently with the main processor and so will refer to a later instruction.

Appendix F

New Features

Summary of Version 3 Improvements

This section is intended as a quick guide to the main additional facilities that Devpac Amiga 3 provides for users who are familiar with version 2.14 and 2.15 of Devpac Amiga. Users of earlier versions of Devpac Amiga 2 should note that a considerable number of features were added during its life time.

We will give an overview of the new features here; for further details you should consult the relevant sections of this manual.

The Editor

The Devpac editor has been greatly enhanced, with multi-file editing, full mouse control, bookmarks, clipboard cut-and-paste, pop-up option menus, extremely fast search and replace, different font sizes being some of the major highlights.

All icons, requesters, gadgets have a Workbench Release 2.0 look and feel, even under 1.3, and a separate Workbench 2.0 version is supplied which takes advantage of the operating system's many new features.

Another new feature of the editor is the ability to open an unlimited number of windows on the same project allowing you to view several sections of a single source file simultaneously giving incredible flexibility.

A macro recording facility is available for teaching the editor any tedious or complex sequence of actions which can then be replayed with a single keystroke.

Full control of assembler options and resident tools is provided via requesters making it extremely simple to configure and customise the operation of GenAm. Assembler options can also be saved to disk for use from the Command Line Interface.

The Assembler

The assembler now fully supports all the 68000 to 68040 and 68332 processors, the 68881/2 maths co-processor and the 68851 MMU. It can also produce S-records in addition to Arniga@ executable and linkable code. To complement the production of S-records we supply an S-record splitter for use with EPROMs that are not the same width as the processor's bus.

The assembler can now generate and process pre-assembled include files. This increases the speed of assembly of programs that use the operating system include files.

LINE and HCLN debug hunks can be generated so that debuggers (including MonAm version 3) can track the source code that corresponds to a given address and vice versa.

The range of options has been extended and options may now be specified by name rather than using cryptic letters. Command line support has been enhanced to allow the setting of labels and otherwise unavailable options. Options - are also read from a default file and this can be created using the editor.

The assembler now gives an indication of where in a line an error was detected. The full range of relational operators are now supported.

Options have been added for listings on pass 1 and for tracing conditional assembly. The use of privileged instructions can now be controlled using the SUPER and USER options.

Further optimisation facilities are provided.

The CARGS and RADIX directives have been added.

\ # may now be used as a synonym for NARGS in macros and the macro.w feature has been added for macros that must generate code on even boundaries.

Default module names are more descriptive.

Compatibility Issues

Most source files should assemble with no changes although the new directive names may clash with existing macro names. Also `.b` may not be used as a local label.

If you are using shell scripts or make files you should note that the standalone version of the assembler is now called GenAm. Because there are now many options that can be used directly on the command line (without a `-` prefix) there is a small chance that this may conflict with the name of one of your source files. If this is the case insert the keyword `FROM` in front of the file name. The options that can be used on the command line without a prefix are summarised at the end of the **Assembler Options** section.

The Debugger

The front panel window display of MonAm can now be organised as you wish. Windows can be split horizontally, vertically and * also stacked in order to extend the number of available work areas. Each stacked window may be locked to an arbitrary expression allowing interactive monitoring of complex data structures.

Any number of source files may be loaded into each window along with any associated line number debugging information such as that output by GenAm. Multi-module programs can thus be single stepped line by line from your original source file. Two powerful new operators are provided which convert a program address into a source line number and locate any part of the program from its position in the source.

The disassembler now recognises all 68000 family processor instructions, including the 68040, maths coprocessor and MMU instructions. The register display has also been updated to show new registers and floating point registers present on the more powerful processors.

Integration

The integration of the package has been further enhanced so that the error commands now work in multiple files and the assembler will read include files from memory without the need to save these to disk. The full range of assembly options is now available via requesters. You can also decide when the assembler and debugger are loaded.

Linker

A much newer version of BLink is now supplied. This is a commercial version of the public domain linker that was supplied with Devpac Amiga 2. The new version is faster, has more options and supports the new base relative features of the AmigaDOS file format.

New Include files

The Commodore assembly language include files for both version 1.3 and 2.04 of the operating system are supplied.

Features added to Devpac Amiga version 2

This section indicates some of features that were added to Devpac Amiga 2 before version 2,14 was released; if you are familiar with an early version of this product you should find it useful.

The Assembler

A number of new optimisations and error checking options were added. The @ character is now allowed in symbols. Labels may be defined on the command line and quoted file names may be used. Local labels ending with a \$ are supported. The extensions for base relative addressing that were originally provide by Lattice C 5.0 are now available.

The Debugger

68020/030 compatibility was added and all memory is checked to see that it exists before it is displayed.

Labels that are embedded in data areas, and full MOVEM register lists are shown when disassembling to disk. The search command is more flexible.

Appendix G

Converting from other Assemblers

Most 68000 assemblers for the Amiga® follow, to one degree or another, the Motorola standard. While the instructions themselves are thankfully standard, the syntax rules for labels, comments and directives can, and do, vary. This Appendix covers the changes most likely to be made when converting programs from another assembler, whether they are your old source files or a program listed in a magazine. It does not attempt to detail the differences in user interfaces or options between the different assemblers.

Amiga® Macro Assembler and MCC Assembler

Almost all source code written for assembly under the AmigaDOS Macro Assembler supplied by Commodore and the Metacomco (MCC) assembler should assemble with little or no change under GenAm. The differences are:

With GenAm, importing constants using XREF and then accessing them as constants (as opposed to relative addresses) may cause warning messages. Many source programs use XREFs for `_LVO` labels. To remove these warnings either:

- Change XREF to XREF.L.
- Use OPT W- to suppress all warnings
- Use OPT T - to suppress type-checking.
- Include the relevant `_lib.i` file and remove the XREF.

The include files supplied with the AmigaDOS Macro Assembler will assemble unchanged by GenAm, but we recommend that you use our pre-assemble feature to speed up your development time.

K-Seka

Colons are not required after labels in GenAm though instructions or directives that start in the label field will need a tab added before them. Several Seka directives default to Byte instead of Word sizes for some reason. Equivalent directives names are:

D=DC; BLK=DS; IF=IFNE; ELSE=ELSEIF; ENDIF-ENDC.

Macro syntax requires ? s to be changed to \ s, except ?0 which should be replaced with \@

Assempro

Most of the Assempro directives are supported by GenAm directly although the macro parameters use % instead of \. Here are some equivalents that Devpac does not support directly:

ALIGN.W=EVEN; ALIGN.L=CNOP 0,4; IBYTES=INCBIN; DEFB=DC.B;
DEFW=DC.W; DEFL=DC.L; DEFM=DC.B

ArgAsm

As ArgAsm was designed to be compatible with Devpac 2 most files should assemble unchanged. Equivalents of the copper instructions that are directly supported by ArgAsm are contained in the macro file that we supply as misc/copper.i. This may be pre-assembled if you wish.

Appendix H

Technical Support

HiSoft Devpac comes with 30 days free technical support, starting from the date of registration; therefore you should send in your registration card quickly. Technical support is available by telephone during our Technical Support Hour, by letter or by fax.

Should you wish to receive extended technical support, please complete the relevant sections on the registration card, indicating whether you would like to take up the *Silver* or the *Gold* service.

In addition to your name, address and postcode (very important for UK customers), we need payment details before we can accept your extended registration. You can pay by credit card (Mastercard, Eurocard, Access, Visa etc.), UK debit card (Switch, Connect etc.), Eurocheque, UK cheque or Postal Order.

You may have already registered another HiSoft product under our Gold or Silver service; in this case, there is no need to fill out the payment section.

Appendix I

Bibliography

This bibliography contains our suggestions for further reading on the subject of the Amiga's operating system, 680x0 assembly language and programming in general. The views expressed are our own and as with all reference books there is no substitute for looking at the books in a good bookshop before making a decision.

We can supply a number of the books listed, especially the AmigaDOS titles - see the order form enclosed with the Devpac package for details on obtaining these books.

Amiga

The AmigaDOS Manual, 3rd Edition

Bantam Books [1991]

ISBN 0-553-35403-5, Bantam Books, London.

Amiga User Interface Style Guide

Commodore-Amiga, Inc. [1991]

ISBN 0-201-57757-7, Addison-Wesley Publishing Company, Inc.

Amiga ROM Kernel Reference Manual: Includes & Autodocs, 3rd Edition

Commodore-Amiga, Inc. [1991]

ISBN 0-201-56773-3, Addison-Wesley Publishing Company, Inc.

Ami a ROM Kernel Reference Manual: Libraries, 3rd Edition

Commodore-Amiga, Inc. [1991]

ISBN 0-201-56774-1, Addison-Wesley Publishing Company, Inc.

Amiga ROM Kernel Reference Manual: Devices, 3rd Edition

Commodore-Amiga, Inc. [1991]

ISBN 0-201-56775-X, Addison-Wesley Publishing Company, Inc.

Amiga Hardware Reference Manual, 3rd Edition

Commodore-Amiga, Inc. [1991]

ISBN 0-201-56776-8, Addison-Wesley Publishing Company, Inc.

Interchange File Format Specification

Commodore-Amiga, Inc. [1988]

CATS, West Chester, PA 19380, USA..

680x0

68000 Assembly Language Programming 2nd Edition

Kane, G., D.Hawkins and L.Leventhal [1987]

ISBN 0-07-881232-1, Osborne/ McGraw-Hill, 2600 Tenth Street, Berkely, CA 94710,- USA.

68000, 68010, 68020 Primer

Kelly-Bootle, Stan and Bob Fowler [1985]

ISBN 067-22405-4, Howard W.Sams & Co., 4300 W.62nd Street, Indianapolis, IN 46268, USA.

M68000 Family Programmer's Reference Manual

Motorola Inc. [1989]

Motorola Semiconductor Products Inc., PO Box 20912 Phoenix, AZ 85036, USA.

Mastering The 68000 Microprocessor

Robinson, Phillip R. [1985]

ISBN 0-8306-1886-4, Tab Books Inc., Blue Ridge Summit, PA 17214, USA.

Microprocessor Systems: A 16-Bit Approach

Eccles, William J. [1985]

ISBN 0-201-11985-4, Addison-Wesley Publishing Company, Reading, MA, USA.

Programming the 68000

Williams, Steve [1985]

ISBN 0-89588-133-0, SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

The MC68000 User's Manual 7th Edition

Motorola Inc. [1989]

ISBN 0-13-567074-8, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

The MC68020 User's Manual 2nd Edition

Motorola Inc. [1985]

ISBN 0-13-566878-6, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

The MC68030 User's Manual

Motorola Inc. [1987]

Motorola Semiconductor Products Inc., PO Box 20912 Phoenix, AZ 85036, USA.

The MC68881/MC68882 User's Manual

Motorola Inc. 11987]

ISBN 0-13-566936-7, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Algorithms & Data Structures

Compilers: Principles, Techniques and Tools

Aho, Alfred V, Ravi Sethi and Jeffrey Q. Ullman [1986]

ISBN 0-201-10194-7, Addison-Wesley Publishing Company, Reading, MA, USA.

Algorithms

Sedgewick, Robert [1988]

ISBN 0-201-06673-4, Addison-Wesley Publishing Company, Reading, MA, USA.

Data Structures and Algorithms

Aho, Alfred V, John E. Hopcroft and Jeffrey D. Ullman [1983]

ISBN 0-201-00023-7, Addison-Wesley Publishing Company, Reading, MA, USA.

Fundamental Algorithms

Knuth, Donald E. [1973]

ISBN 0-201-03809-9, Addison-Wesley Publishing Company, Reading, MA, USA.

Seminumerical Algorithms

Knuth, Donald E. [1981]

ISBN 0-201-03822-9, Addison-Wesley Publishing Company, Reading, MA, USA.

Sorting and Searching

Knuth, Donald E. [1973]

ISBN 0-201-03803-X, Addison-Wesley Publishing Company, Reading, MA, USA.

Index

- = directive 113
- __G2, reserved symbol 109
- __LK, reserved symbol 109
- __MERGED, linker 189
- __RESBASE, __RESLEN, linker 195
- __RS, reserved symbol 115

■ A

- absolute expressions 80, 83
- ADDQ/SUBQ, optimising 100
- address ADD/SUB, optimising 100
- address shortening, optimising 100
- addressing modes 83
- alignment, EVEN directive 91, 106
- ALV, warnings, linker 192
- Amiga macro assembler 257
- ArgAsm 258
- assembler
 - addressing modes 83
 - character constants 81, 82
 - command line 72
 - command-line
 - case-insensitive labels, --C 72
 - defaults 73
 - defining symbols, -E 74
 - extended debugging, -X 73
 - Include directory, -I 73
 - linkable code, -L 73
 - list symbol table, -S 73
 - listing file, -P 73
 - listing tab size, -T 73
 - options file, WITH 72
 - other options, -V 73
 - output file, -O 73
 - output file, TO 72
 - pass 1 listing, -z 73
 - pause after assembly, -Q 73
 - pre-assembled files, -I 72
 - S-record output, -L6 73
 - Setting labels, -E 72

- slow mode, -M 73
- source file name 72
- standard debug, -D 72
- suppress binary output, --B 72
- comment 79
- conditional assembly 119
- conditionals
 - end, ENDC 122
 - if defined, IFD 121
 - if equal, IFEQ 120
 - if equivalent, IFC 121
 - if greater than or equal, IFGE 120
 - if greater than, IFGT 120
 - if less than or equal, IFLE 120
 - if less than, IFLT 120
 - if not defined, IFND 121
 - if not equivalent, IFNC 121
 - one line if, IIF 120
 - toggle, ELSEIF & ELSE 122
 - trace option 98
- data type 81
- DEBUG option 72
- dialog box 65
- directives 93
 - =, equate label 113
 - CARGS, define parameter offsets 116
 - CNOP, conditional alignment 106
 - conditional assembly 119
 - DC, define constants 106
 - DCB, define constant block 107
 - DS, define space 107
 - ELSE, toggle conditional assembly 122
 - ELSEIF, toggle conditional assembly 122
 - END, end assembly 93
 - ENDC, end conditional assembly 122
 - ENDM, end macro definition 122
 - ENDR, end repeat loop 110

- EQU, equate label 113
- EQUR, equate register 113
- EVEN, alignment 106
- FAIL, user error 107
- FEQU, equate float constant 117
- FOPT, float options 118
- FORMAT, control listing 112
- IFC, if equivalent 121
- IFD, if defined 121
- IFEQ, if equal 120
- IFGE, if greater than or equal 120
- IFGT, if greater than 120
- IFLE, if less than or equal 120
- IFLT, if less than 120
- IFNC, if not equivalent 121
- IFND, if not defined 121
- IFNE, if not equal 120
- IIF, one line if 120
- INCBIN, include binary 95
- INCDIR, include directory 95
- INCLUDE, include source 93
- LIST, enable listing 110
- LISTCHAR, output control sequence 112
- LLEN, set line width 111
- MACRO, define macro 122
- MEXIT, exit macro invocation 122
- NOLIST, disable listing 111
- OFFSET, start offset section 115
- OPT, set option 95
- ORG
- S-records 136
- output 131
- OUTPUT, output filename 108
- PAGE, page throw 112
- PLEN, set page length 111
- RADIX, default number base 108
- REG, register list 114
- REPT, repeat loop 110
- RS, reserve space 114
- RSRESET, reset RS counter 115
- SECTION
- S-records 136
- SET, temporary equate 113
- SPC, output space 112
- SUBTTL., set sub-title 112
- summary 136
- TTL, set title 112
- executable code 66
- executable files 130
- expressions 80
 - absolute 80, 83
 - relative 80, 83
- file types 75
- floating point 82
- include directories 68
- instruction set 91
 - extensions 91
 - branches 92
 - condition codes 91
 - DBRA 92
 - MOVE CCR 92
- invoking 65
- label 78
- labels
 - equate register, EQUR 113
 - equate, EQU, = 113
 - float equate, FEQU 117
 - parameter offsets, CARGS 116
 - register list, REG 114
 - reserve space, RS 114
 - reset RS counter, RSRESET 115
 - start offset section, OFFSET 115
 - temporary equate, SET 113
- linkable code 66
- linkable files 130
- listing 67
 - control 112
 - disable 111
 - enable 110
 - line width 111
 - output control sequence 112
 - output space 112
 - page length 111
 - page throw 112
 - sub-title 112
 - tab size 67, 73

- title 112
- local labels 78, 89
- macros 122
 - define macro, MACRO 122
 - end macro definition, ENDM 122
 - exit macro invocation, MEXIT 122
 - expansion in listing 124
 - labels, unique 123
 - parameters 123
 - size 123
- memory requirements 66
- mnemonic 79
- multiple includes 66
- numbers 81
- opcode 79
- operand 79
- operators 81
- options 65
 - automatic PC mode 102
 - base displacements 69, 96
 - branch control 97
 - case-sensitivity
 - CASE 97
 - command line 72
 - check evenness 101
 - check immediate 69, 101
 - co-processor 96
 - debugging information 67, 98
 - command line 72
 - defaults 103
 - define variables 67
 - dependent casing 69
 - disable type checking 102
 - export debug 98
 - fast 66
 - file name 68
 - generate symbol table file 71
 - include directories 68
 - include suppression 102
 - independent casing 69
 - insensitive casing 69
 - line debug 98
 - listing 67
 - listing control 97
 - local labels 89
 - local symbol lead in character 103
 - macro expansions 97
 - main file 67
 - maths co-processor 69
 - MMU 69
 - multiple include suppression 102
 - multiple includes 66
 - OPT directive 95
 - optimisations 70, 99
 - ADDQ/SUBQ 100
 - address ADD/SUB 100
 - address shortening 100
 - backward branches 99
 - base displacements 100
 - branch-to-next 100
 - disable all 100
 - enable all 100
 - forward branches 100
 - LEA to. ADD/SUB 100
 - MOVEQ 100
 - outer displacements 100
 - register indirect 99
 - short-word addressing 100
 - outer displacements 69, 96
 - pass one listing 66, 98
 - position independent checks 69, 101
 - processor selection, P 96
 - program type 66
 - sensitive casing 69
 - significance 97
 - slow 66
 - source checking 101
 - summary 103
 - supervisor instructions 103
 - suppress warnings 102
 - symbol case 69
 - symbol table 98
 - symbol table list 66
 - trace conditionals 66, 98
 - type checking 102
 - use symbol table file 68

- variables 67
- warning suppression 102
- output 131
 - S-records 135
- output filenames 76
- output formats 130
- pausing 74
- periods 90
- position-independent code 77
- processor selection, MACHINE 108
- relocatable code 77
- return codes 75
- S-records 66, 73, 75, 130, 135
- settings 51
- stand-alone see assembler,
 - command line
- symbol table files 68, 71
- syntax 78
- to memory 71
- type combinations 83
- using from the editor 55
- warnings 220
- word alignment 91
- assembler version, __G2 109
- assembler, error messages 213
- Assempro 258
- assigns
 - BLINKWITH variable 194
- automatic PC mode, option 102
- AUTOPC option 102

■ **B**

- backspace key 35
- backups
 - editor 44
 - in the editor 48
 - master disk 14
- backward branches, optimising 99
- base displacements 69
- optimising 100
- option 96
- batch mode, linker 192
- BDL option 96
- BDW option 96

- Bibliography 261
- binary numbers 81,108
- Blink The Linker 187
- BLINKWITH assign variable 194
- block
 - delete 35
 - goto start/end 37
 - marking 36
- blocks 36
- bookmarks, editor 34
- bookmarks, goto, editor 42
- books, technical 17, 261
- BRA.L 97
- branch control, option 97
- branch size 69
- branch-to-next, optimising 100
- branches, extension 92
- BRB option 97
- breakpoint, debugger (see debugger, breakpoint)
- BRL option 97
- BRS option 97
- BRW option 97
- buffer size, linker 192

■ **C**

- CARGS directive 116
- case dependency, editor 40
- CASE option 97
- case-sensitivity, option 97
- character constants 81, 82, 108
- check evenness, option 69, 101
- check immediate, option 69, 101
- CLI, using it 233
- clipboard, editor 37
- CNOP directive 106
- co-processor, option 96
- command file, linker 193
- command line
 - assembler 72
 - S-record splitter 205
- commands, editor 19
- comment field 79
- compatibility

- with other assemblers 257
- with Devpac 2 253
- condition codes, extension 91
- conditional alignment, CNOP 106
- conditional assembly see assembler
- conditionals, trace option 98
- control listing, FORMAT 112
- control options, OPT 95
- copy block, editor 37, 38
- copy memory, debugger 177
- cross referencing, linker 191
- cursor keys
 - debugger 158
 - in the editor 31
 - with keyboard modifiers 31
- cursor, use in the editor 31
- cut block, editor 37
- **D**
- data types, assembler 81
- DBRA, extension 92
- DC directive 106
- DCB directive 107
- DEBUG
 - command line option 72
 - suppression with linker 189
- debugger
 - abort 170
 - address, set 159
 - breakpoint 166
 - conditional 167
 - count 166
 - kill 168
 - permanent 166
 - remove 168
 - set 164, 167, 168
 - show 168
 - simple 166
 - stop 166
 - command summary 179
 - copy memory 177
 - current drive 179
 - cursor keys 158
 - disassemble 178
 - disassembly window 156
 - edit 160
 - exceptions
 - description 140
 - executing programs 172
 - expressions 151
 - fill memory 178
 - find 173
 - hints 182
 - history 169
 - labels 153
 - list 177
 - load binary 171
 - load program 171
 - load source 172
 - lock window 160
 - memory window 158
 - MonAm 139
 - print window 161
 - register
 - set 165
 - window 155
 - registers 153
 - running program 173
 - go 173
 - running programs 172
 - save binary 171
 - screen switching 165
 - search memory 173
 - settings 175
 - single step 172
 - skip 173
 - split window 161
 - symbols 153
 - terminate 170
 - trace 173
 - user screen 165
 - windows 154
 - commands 159
 - disassembly 156
 - edit 160
 - lock 160
 - memory 158

- print register 155
- source code 158
- split 161
- type 161
- zoom 162
- zoom window 162
- debugging information 67,98
- decimal numbers 81
- define constant block, DCB 107
- define constants, DC 106
- define macro, MACRO 122
- define space, DS 107
- MACRO, define macro 122
- defined conditional, IFD 121
- defining symbols, command line 74
- defining symbols, linker 192
- delete block, editor 38
- deleting files, editor 45
- deletion of text, editor 34
- dependent casing, symbol 69
- directives
 - =, equate label 113
 - assembler 93
 - CARGS, define parameter offsets 116
 - CNOP, conditional alignment 106
 - DC, define constants 106
 - DCB, define constant block 107
 - DS, define space 107
 - ELSE, toggle conditional assembly 122
 - ELSEIF, toggle conditional assembly 122
 - END, end assembly 93
 - ENDC, end conditional assembly 122
 - ENDM, end macro definition 122
 - ENDR, end repeat loop 110
 - EQU, equate label 113
 - EQUR, equate register 113
 - EVEN, alignment 106
 - FAIL, user error 107
 - FEQU, float equate 117
 - FOPT, float options 118
 - FORMAT, control listing 112
 - IFC, if equivalent 121
 - IFD, if defined 121
 - IFEQ, if equal 120
 - IFGE, if greater than or equal 120
 - IFGT, greater than 120 161
 - IFLE, f less than or equal 120
 - IFLT, if less than 1.20
 - IFNC, if not equivalent 121
 - IFND, if not defined 1.21
 - IFNE, if not equal 120
 - IIF*, one line if 120
 - INCBIN, include binary 95
 - INCDIR, include directory 95
 - INCLUDE, include source 93
 - LIST, enable listing 110
 - LISTCHAR, output control sequence 112
 - LLEN, set line width 11.1
 - MACHINE selection 108
 - MEXIT, exit macro invocation 122
 - NOLIST, disable listing 111
 - OFFSET, start offset section 115
 - OPT, control options 95
 - ORG
 - s-records 136
 - OUTPUT, output filename 108
 - PAGE, page throw 112
 - PLEN, set page length 111
 - RADIX, default number base 108
 - REG, register list 114
 - REPT, repeat loop 110
 - RS, reserve space 114
 - RSRESET, reset RS counter 115
 - SECTION
 - S-records 136
 - SET, temporary equate 113
 - SPC, output space 112
 - SUBTTL, set sub-title 112
 - summary 136
 - TTL, set title 112
- disable all, optimising 100

- disable listing, NOLIST 111
- disable type checking, option 102
- disk contents 11
- disks
 - backup 14
- double precision 246
- DS directive 107

■ **E**

- editor 17

- automatic indent 48
 - backups 44, 48
 - block commands 36
 - clipboard 37
 - copy block 37,38
 - cut block 37
 - delete block 38
 - marking a block 36
 - paste block 37, 38
 - print block 39
 - save block 39
 - bookmarks 34
 - centre window 54
 - clipboard 37
 - commands 19
 - cursor keys 31
 - default settings 19
 - delete commands 36
 - delete file 45
 - deleting text 34
 - file requester 23
 - find see search
 - icons 19
 - inserting text 44
 - keyboard shortcut summary 60
 - keyboard shortcuts 20
 - load text 42
 - loading settings 51
 - macros 45
 - making backups 44
 - marks 34
 - menus 26
 - numeric pad 48
 - printing 53

- project 28
 - replace all 41
 - replacing 39
 - requesters and gadgets 20
 - resident tools 56
 - save text 43
 - search
 - case dependency 40
 - control characters 41
 - next 41
 - special characters 41
 - tab 41
 - searching 39
 - settings 19, 46
 - saving 50
 - starting up 19
 - sub-menus 26
 - tab size 47
 - the window 27
 - undo line 31
 - using fonts 54
 - using the assembler 55
 - using the cursor 31
 - using workbench icons 43
 - window layout 30
 - windows, switching 29
 - workbench icons 49
- ELSE directive 122
- ELSEIF directive 122
- enable all, optimising 100
- enable listing, LIST 110
- end conditional assembly, ENDC 122
- END directive 93
- end macro definition, ENDM 122
- end repeat loop, ENDR 110
- ENDC directive 122
- ENDM directive 122
- ENDR directive 110
- EPROMs 205
- EQU directive 113
- equal conditional, IFEQ 120
- equate float label, FEQU 117

- equate label, EQU, := 113
- equate register, EQU 113
- equivalent conditional, IFC 121
- EQU 113
- error messages, AmigaDOS 209
- error messages, assembler 213
- EVEN directive 106
- executable code 66
- executable files 130
- executing programs, debugger 172
- exit macro invocation, MEXIT 122
- exponent 245, 246, 247
- export debug, option 98
- expressions
 - assembler 80
 - imports 134
 - inter-section references 134
- expressions, debugger 151
- extensions
 - branches 92
 - condition codes 91
 - DBRA 92
 - MOVE CCR 92

■ **F**

- FAIL directive 107
- fast assembly 66
- FD2LVO 206
- FEQU directive 117
- file name field width, linker 191
- file requester 23
- find next, editor 41
- find, debugger 173
- find, editor see editor, searching
- float equate, FEQU 117
- float options, FEQU 118
- floating point 82
 - double precision format 246
 - extended precision format 245
 - options, FEQU 118
 - packed decimal format 247
 - single-precision format 246
- floating point co-processor 245
- floating point control register 248

- floating point instruction address
- register register 250
- floating point status register 249
- fonts 54
- FOPT directive 118
- FORMAT directive 112
- forward branches, optimising 100
- from files, linker 189

■ **G**

- gadgets
 - depth 29
 - in the editor 20
- GenAm, Stand-alone assembler 71
- GenAm.opts 72
- GENSEYM files
 - generating 71
 - using 68
- GENSEYM option 94
- goto bookmark, editor 42
- goto line, editor 41
- greater than conditional, IFGT 120
- greater than or equal conditional, IFGE 120

■ **H**

- HCLN debug, option 99
- headers, pre-assembled files 94
- hexadecimal numbers 81, 108
- hints
 - debugger 182
- history
 - debugger 169
- hunk field width, linker 191

■ **I**

- icons 49
 - in the editor 43
- IFC directive 121
- IFD directive 121
- IFEQ directive 120
- IFGE directive 120
- IFGT directive 120
- IFLE directive 120
- IFLT directive 120

- IFNC directive 121
- IFND directive 121
- IFNE directive 120
- ignore errors, linker 192
- IIF directive 120
- imports in expressions 134
- INCBIN directive 95
- INCDIR directive 95
- include binary, INCBIN 95
- INCLUDE directive 93
- include directory
 - command-line option 73
 - INCDIR directive 95
- include files
 - multiple 66
 - search path 68
- include source, INCLUDE 93
- include suppression, option 102
- INCONCE option 102
- independent casing, symbol 69
- infinity 246, 247
- insensitive casing, symbol 69
- inserting text, editor 44
- instruction set 91
- inter-section references in
 - expressions 134

■ **K**

- K-Seka 258
- keyboard shortcuts
 - editor 20, 60
- keys
 - backspace 35
 - del 35
- keywords, linker 189, 190

■ **L**

- labels, debugger 153
- labels, macro 123
- labels, valid 78
- LEA to ADD/SUB, optimising 100
- less than conditional, IFLT 120
- less than or equal conditional,
 - IFLE 120

- libraries 223
- libraries, linker 189
- line
 - delete 35
 - undelete 35
- LINE debug, option 98
- line length, linker 192
- line, goto editor 41
- linkable code 130
 - assemble from editor 66
 - command-line option 73
- Linker 187
 - __LinkerDB 195
 - __MERGED 189
 - __RESBASE, __RESLEN 195
 - ALV warnings 192
 - batch mode 192
 - Blink 187
 - buffer size 192
 - columns, number of 191
 - command file 193
 - cross referencing 191
 - debug suppression 189
 - defining symbols 192
 - Errors 197
 - file name field width 191
 - from files 189
 - hunk field width 191
 - ignore errors 192
 - keywords
 - ADDSYM 189
 - BATCH 192
 - BUFSIZE 192
 - CHIP 189
 - DEFINE 192
 - FANCY 191
 - FAST 189
 - HEIGHT 191
 - HWIDTH 191
 - IGNORE 192
 - INDENT 192
 - LIB 189
 - LIBRARY 189 -

- MAP 191
- MAXHUNK 189
- ND 189
- NOAVLS 192
- NODEBUG 189
- PLAIN 192
- PRELINK 190
- PWIDTH 192
- QUIET 193
- SMALLCODE 190
- SMALLDATA 190
- SWIDTH 192
- TO 190
- VER 193
- VERBOSE 193
- VERIFY 193
- WIDTH 192
- WITH 193
- XREF 191
- keywords.FWIDTH 191
- libraries 189
- line length 192
- listing indent 192
- log file name 193
- map file 191
- number of columns 191
- output filename 190
- pre-linking 190
- program width 192
- quiet mode 193
- symbol width 192
- symbols 189
- verbose mode 193
- width, file name field 191
- with file 193
- LIST directive 110
- LIST1 option 98
- LISTCHAR directive 112
- listing
 - filename 67
 - indent, linker 192
 - macros 124
 - options in assembler 97

- symbol table 66
- tab size 73
- LLEN directive 111
- load binary, debugger 171
- load program, debugger 171
- load source, debugger 172
- local labels, assembler 78, 89
- local symbol lead in character,
 - option 103
- LOCALDOT option 103
- LOCALU option 103
- log file name, linker 193
- **M**
- MACHINE directive 108
- macro arguments, number of,
 - NARG 123
- macro definition, MACRO 122
- MACRO directive 122
- macro expansions, option 97, 124
- macros 45, see assembler, macros
- mantissa 245, 246, 247
- manual, how to use 1
- map file, linker 191
- mark block, editor 36
- marks, editor 34
- master disk, backup 14
- maths co-processor 69, 245
- memory requirements 4,66
- menus, in the editor 26
- MetaComCo macro assembler 257
- MEX option 97
- MEXIT directive 122
- MMU 69
- mnemonic field 79
- MonAm (see debugger) 139
- MOVE CCR, extension 92
- MOVEQ, optimising 100
- multiple include suppression,
 - option 102
- **N**
- NaN see Not-A-Number
- NARG, reserved symbol 123

- new features 251
- NOCASE
 - command line option 72
 - option 97
- NOLIST directive 111
- NOLIST1 option 98
- NOMEX option 97
- NOSYMTAB option 98
- not defined conditional, IFND 121
- not equal conditional, IFNE 120
- not equivalent conditional, IFNC 121
- Not-A-Number 246, 247
- NOTRACEIF option 98
- number base, default RADIX 108
- number of columns, linker 191
- numbers, assembler 81
- ●
- octal numbers 81,108
- ODL option 96
- ODW option 96
- OFFSET directive 115
- OLD option 97
- one line if, IIF 120
- opcode field 79
- operand field 79
- operators
 - debugger 151
- operators, assembler 81
- OPT directive 95
- optimising option 70, 99
 - ADDQ/SUBQ 100
 - address ADD/SUB 100
 - address shortening 100
 - backward branches 99
 - base displacements 100
 - branch-to-next 100
 - disable all 100
 - enable all 100
 - forward branches 100
 - LEA to ADD/SUB 100
 - MOVEQ 100
 - outer displacements 100
 - register indirect 99
 - short word addressing 100
 - outer displacements 96
 - pass one listing 98
 - position independent checks 69, 101
 - processor 96
 - significance 97
 - source checking 101
- register indirect 99
- short-word addressing 100
- options
 - automatic PC mode 102
 - base displacements 96
 - branch control 97
 - case-sensitivity 97
 - check evenness 69, 101
 - check immediate 69, 101
 - co-processor 96
 - debugging information 98
 - defaults 103
 - dialog box 65
 - disable type checking 102
 - export debug 98
 - include suppression 102
 - line debug 98
 - listing control 97
 - local symbol lead in character 103
 - macro expansions 97
 - multiple include suppression 102
 - OPT directive 95
 - optimisations 70, 99
- ADDQ/SUBQ 100
 - address ADD/SUB 100
 - address shortening 100
 - backward branches 99
 - base displacements 100
 - branch-to-next 100
 - disable all 100
 - enable all 100
 - forward branches 100
 - LEA to ADD/SUB 100
 - MOVEQ 100
 - outer displacements 100
 - register indirect 99
 - short word addressing 100
 - outer displacements 96
 - pass one listing 98
 - position independent checks 69, 101
 - processor 96
 - significance 97
 - source checking 101

- summary 103
- supervisor instructions 103
- suppress warnings 102
- symbol table 98
- trace conditionals 98
- type checking 102
- warnings suppression 102

ORG directive

- S-record format 136

outer displacements 69

- optimising 100
- option 96

output control sequence,
LISTCHAR 112

OUTPUT directive 108

output directives 131

output filename 68

- linker 190

OUTPUT 108

output format, __LK 109

output formats 130

output space, SPC 112

■ P

P= option 96

packed decimal format 247

PAGE directive 112

page throw, PAGE 112

parameters, macro 123

pass one listing

- command line option 73
- from editor 66
- option 98

paste block, editor 37, 38

pausing, assembler 74

periods, assembler 90

PLEN directive 111

position independent checks,
option 69,101

position-independent code 77

pre-assembled files 94

- command-line option 72
- generating 71
- using 68

pre-linking, linker 190

precedence, assembler operators 81

print block, editor 39

printing

- from the editor 53

processor type 68

processor, option 96

program type 66

program width, linker 192

project 28

- read only 54

pseudo-mnemonics 93

■ Q

quiet mode

- linker 193

■ R

RADIX directive 108

README File 15

REG directive 114

register indirect, optimising 99

register list, REG 114

registers, debugger 153

registration card 15

relative expressions 80, 83

relocatable code 77

relocation information 77

repeat loop, REPT 110

replace all, editor 41

replacing, editor 39

REPT directive 110

requesters

- confirmation 25
- file 23
- in the editor 20

reserve space, RS 114

reserved symbols

- __G2, assembler version 109
- __LK, __RS, __G2 79
- __LK, output format 109
- __RS, RS counter 115
- linker 195
- NARG, number of macro arguments

- 123
- reset RS counter, RSRESET 115
- return codes, assembler 75
- RS counter, RS 115
- RS directive 114
- RSRESET directive 115
- Running program
 - debugger 173
- Running programs, debugger 172

■ S

- S-record format 130, 135
 - assemble from editor 66
 - assembler 135
 - command-line option 73
 - definition 135
 - ORG directive 136
 - SECTION directive 136
- save binary, debugger 171
- save block, editor 39
- saving text, editor 44
- screen switching, debugger 165
- search 39
 - case dependency 40
 - next 41
 - replace all 41
 - replacing 39
- search, debugger 173
- section (See hunk)
- SECTION directive
 - S-record format 136
- sensitive casing, symbol 69
- SET directive 113
- set line width, LLEN 111
- set page length, PLEN 111
- set sub-title, SUBTTL 112
- set title, TTL 112
- settings
 - assembler 51
 - debugger (see debugger, preferences)
 - editor 46
- short-word addressing, optimising 100

- significance, option 97
- single precision 246
- single step, debugger 172
- skip, debugger 173
- slow assembly 66
- source checking, options 101
- source code, debugger 158
- SPC directive 112
- stand-alone assembler 71
- SUBTTL directive 112
- SUPER option 103
- supervisor instructions, option 103
- symbol case 69
- symbol table
 - assembler option 98
 - list 66
- symbol width, linker 192
- symbols
 - debugger 153
 - defining to linker 192
 - linker 189
- SYMTAB option 98
- syntax, assembler 78

■ T

- tab key 33
- tab size 47
- tab size, assembler listing 67
- technical support 259
- temporary equate, SET 113
- terminate, debugger 170
- text
 - insert, editor 44
 - load, editor 42
 - save, editor 43
- toggle conditional assembly,
 - ELSEIF & ELSE 122
- trace conditionals, option 66, 98
- trace, debugger 173
- TRACEIF option 98
- TTL directive 112
- tutorial, quick 6
- type checking, option 102
- type combinations, assembler 83

TYPE option 102

typography 4

■ **U**

user error, FAIL 107

USER option 103

■ **V**

variables, defining from editor 67

verbose mode, linker 193

■ **W**

WARN option 102

warnings, assembler 220

width, file name field in linker 191

windows

- debugger (see debugger, windows)

- editor, centre 54

- layout 30

- switching editor windows 29

- the editor 27

with file, linker 193

word alignment 91

■ **Z**

zoom window, debugger 162



HiSoft Systems, The Old School, Greenfield, Bedford, MK45 5DE, UK
tel +44 (0) 1525 718181 **fax** +44 (0) 1525 713716 **web** www.hisoft.co.uk